1-1-2017

Automated Refinement Of Hierarchical Object Graphs

Mohammad Ebrahim Khalaj
Wayne State University,

AUTOMATED REFINEMENT OF HIERARCHICAL OBJECT GRAPHS

by

MOHAMMAD EBRAHIM KHALAJ

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2017

MAJOR: COMPUTER SCIENCE

Approved By:

_____

Advisor Date

_____

_____

_____

_____

DEDICATION

To all the hard-working people who try everyday to reach their goals, without doubting themselves.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Marwan Abi-Antoun, for his infectious enthusiasm for my research project. We were both excited to be finally making major inroads on the inference of ownership type qualifiers, which boosted several analyses and applications developed in our research lab, which require those qualifiers to be present and, until now, had to be added manually. He not only kept a close eye on me and my work to ensure its high quality, but also taught me how to become a more responsible person. My journey on this challenging technical problem would have been ten times harder without a hands-on advisor who stepped in and helped me overcome one obstacle after the next on this project.

Also, I would like to thank my other dissertation committee members. Dr. Vaclav Rajlich, for reading my dissertation carefully and helping me highlight the most important parts of my work. Dr. Nathan Fisher, for his warm feedback on how to position my work with respect to closely related work. And Dr. Andrian Marcus with his pointed questions that made me emphasize the relevance and impact of my work.

I want to thank current and alumni of the SEVERE research group: Dr. Radu Vanciu for patiently answering my many questions starting from my first year as a Ph.D. student; Sumukhi Chandrashekar for sharing parts of our Ph.D. journey with the same hopes and fears; Yibin Wang for carefully reading several paper drafts and giving me very helpful comments; Mohammad Anamul Haque, Wesley Trescott and Andrew Giang for using my tool and giving me usability feedback. I also want to thank Dr. Laura Moreno and Oscar Chaparro for being good lab mates in my first year in graduate school. Many thanks to the Department of Computer Science and its kind and helpful Chair Dr. Loren Schwiebert for their support.

Moreover, I would like to thank my family. Having me leave my home country to pursue my Ph.D. was hard on my entire family, but more so on my parents. But I knew they would

all be close to me, in my heart, during my journey. I cannot thank enough my father and my mother, Hojjatollah and Mehry, for their tender love and support, also my brother Dr. Mohammadreza and my beautiful sisters Maryam and Monir for believing in me. Talking to them and seeing them through video calls helped me deal with the fact that I missed them so much.

My warmest and deepest thanks go to my wife Nona for her unconditional love and support. She worked as hard as me on my tough days in graduate school. She was supportive of me working long hours and weekends. She kept me motivated, even when it was very hard for me to stay the course.

Now that I am at the finish line of my Ph.D., when I look back, I see a great experience, and this makes me happy that I had the chance to practice it. This was a life changing journey for me, and I feel that it made me a more mature person. Maybe I am done with my Ph.D., but I am not done with the other challenges of life, and I look forward to tackling them too.

Funding. This work was supported in part by Wayne State University, and in part by National Security Agency lablet contract #H98230-14-C-0140. The views and conclusions contained herein are my own and should not be interpreted as representing the policies or opinions of Wayne State University or any of the sponsors of this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

In order to evolve object-oriented code, developers must understand its run-time structure in terms of objects and their relations, as well as they must understand the code structure dealing with source files, classes and packages. For object-oriented code, it is hard to understand the run-time structure from looking at the code. Moreover, snapshots of runtime heaps do not convey design intent, since they reflect one execution, not all possible ones. Thus, abstractions of the run-time structure such as points-to graphs or abstract object graphs can be highly complementary to diagrams of the code structure such as class diagrams as well as views of runtime heaps [24]. This dissertation is about statically extracted abstractions, so we use *object* to mean abstract object rather than runtime object and *object graph* to mean abstract object graph.

Unfortunately, tools for object graphs are still immature, compared to tools for the code structure. One reason is that extracting these object graphs from code is difficult. Ideally, the object graph must be sound and show all possible objects and possible communication between objects. Achieving soundness requires static analysis which, without additional information, extracts large, non-hierarchical object graphs that do not convey much abstraction.

One way to make object graphs convey abstraction is to use object hierarchy, where important objects of the application appear near the top of the hierarchy and data structures are further down. One way to supply information about object hierarchy is to use qualifiers that implement several type systems with varying degrees of expressiveness, which have been proposed by the programming languages community over the past 15 years [15, 18, 11]. However, until recent work on inference of ownership type qualifiers [21, 17, 37], developers had to supply these qualifiers, which is a significant burden, and is one of the reasons that ownership type systems have not been adopted widely. Moreover, it is hard for most developers, from looking at the code, to decide what type qualifiers to add, because it is hard

to visualize the object structure that the qualifiers describe. In fact, most research papers that describe ownership type systems include several diagrams, typically hand-drawn, to clarify the object structures. When using ownership type systems, developers must add most of the qualifiers upfront, before they can visualize the object graph. Then, based on the visualization, they refine the qualifiers. In other words, to add precise qualifiers that express design intent, the developers rely on a depiction or a mental model of the object graph. Even when not using object graphs, the inference must not be only batch-oriented: developers must be able to guide it, for example, by adding some qualifiers in the code and letting the analysis infer the remaining ones. Without any developer guidance, a batch-oriented approach often infers abstractions that developers do not recognize. This guidance, however, must not involve a significant manual burden.

By comparison, many tools to extract diagrams of the code structure [2] allow developers to drag-and-drop classes and interfaces onto a canvas, choose the relationships (inheritance, association, etc.) to include, and get a diagram. We incorporate the same insight and make the process of extracting hierarchical object graphs more interactive without requiring developers to switch between adding qualifiers to the code and extracting object graphs.

1.1 Applications of Object Graphs

In this section we discuss some application of object graphs.

Program comprehension. Diagrams of software structure are useful during software evolution. Class diagrams show the static code structure. Object graphs approximate the runtime structure in terms of objects. So giving developers these complementary diagrams is likely to help with program comprehension. Ammar and Abi-Antoun [13] conduct a user study with developers to evaluate the effectiveness of object graphs. In the user study, one group of developers are given only the class diagrams, and another group are given both class diagrams and object graphs. The results of the study show that when developers have both types of diagrams, they perform code modification tasks more effectively, e.g., by browsing less irrelevant code. Object graphs also have pedagogical applications to help

novice developers understand some structural object-oriented design patterns and explain shallow or deep cloning [8].

Reasoning about security. Another application of object graphs is to reason about security at a higher level of abstraction than lines of code. Vanciu and Abi-Antoun [34] propose an approach to find architectural flaws in the code, by assigning security properties to objects then writing constraints on the object graph. For example, developers set the property of an object that contains confidential information (like a password) to be true. They write constraints or build queries [9] to search the object graph to ensure that confidential information does not flow to an object that is set to be untrusted. Moreover, using the same idea, the developers analyze the possibility of tampering, by ensuring tainted information does not flow to trusted objects.

Conformance analysis. An object graph can be abstracted into a runtime architecture. Using a conformance analysis, architects can compare the as-built architecture extracted from the code, to a target, documented architecture and measure their structural conformance. This conformance analysis can find interesting architectural differences between the as-built system and its target architecture such as when the documented diagram is missing important communication [5].

Impact analysis. Another application of object graphs is for impact analysis, which is estimating the changes of the code for a change request [14]. A static analysis may mine a hierarchical object graph and its edges to rank dependencies such as the most important classes related to a class, or the important class behind an interface [10]. Using statically extracted dependencies is usually more complicated for object oriented programs due to sub-classing, interfaces, aliasing, collections among others. Using hierarchical object graphs during impact analysis leads to exploring less irrelevant code, since it traverses more precise dependencies compared to dependencies from the program's abstract syntax tree.

We list just a few of the possible applications of object graphs. There are others, dealing with distributing nodes [30]. The main reason for the lack of adoption of object graphs is the

immaturity of the today's tools. Prior to this work, extracting meaningful object graphs is hard and overwhelming for developers. This dissertation addresses the problem of extracting object graphs by proposing an approach that enables developers to automatically extract an object graph, or to refine an initial flat object graph to further to express their design intent.

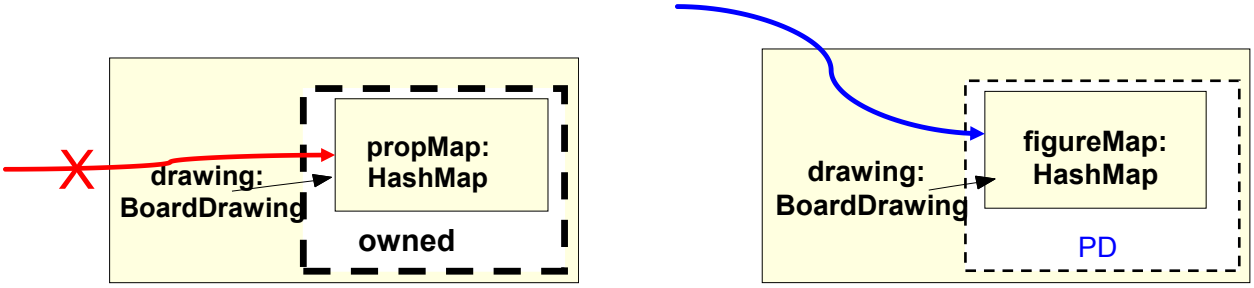
1.2 Benefits of Ownership Qualifiers in the Code

Although adding ownership qualifiers to the code manually is overwhelming for developers, having them in the code has its own benefits. Ownership qualifiers can improve code quality, by identifying cases of unwanted aliasing, or exposing shallow versus deep cloning [8]. Using ownership qualifiers, developers can express their design intent, by identifying strictly encapsulated objects, or objects that are logically contained in the other objects [4]. Also they can make objects peers or make objects access each other through ownership parameters. Another benefit of ownership qualifiers in the code is to make more explicit in the implementation the design patterns in use [27]. However, not all ownership types are flexible enough to express the common design patterns. For example, the Observer design pattern is challenging to express [29].

1.3 Motivation: Refining Object Graphs Without Changing Qualifiers

Today, the most significant limitation of extracting object graphs is the effort involved in adding qualifiers, measured at around 1 hour/KLOC [6]. The effort is due to the high overhead associated with inserting ownership qualifiers into the code, then refining the qualifiers both to get them to type-check, and to ensure that the qualifiers capture hierarchy in a way such that the extracted object graph reflects a global hierarchy that matches the developers' design intent.

A related issue is bootstrapping the process of extracting an object graph. Today, developers add most of the qualifiers, type-check the qualifiers and fix all of the high-priority



(a) Strict encapsulation (owned-by). (b) Logical containment (part-of).

Figure 1.1: Two ways to create hierarchy in an object graph. Box nesting indicates ownership or containment. The object `propMap` is in the `owned` domain of the object `drawing`, and inaccessible from the outside. The object `figureMap` is inside the domain `PD` and accessible to the outside.

warnings produced by the type-checker before they can extract an initial object graph. Only then, based on visualizing the extracted object graph, they iterate the process of refining the qualifiers. In other words, to add better qualifiers, the developers rely on the knowledge provided by the object graph.

Another issue is that the process of refining the extracted object graph is currently somewhat awkward: developers must notice where the object graph does not match their design intent and identify the cases where there are incorrect qualifiers in the code, rather than a mismatch between the as-implemented system and the developer's design intent. If the issue is in the qualifiers, the developers have to change the qualifiers consistently to reflect the correct design intent, then re-run the static analysis to extract the object graph.

Today, these issues make the process of extracting and refining object graphs tedious and time-consuming, and make object graphs less useful to developers. This dissertation addresses these issues by extracting an initial object graph automatically, then allowing developers to directly and interactively refine the extracted object graph to make it convey their design intent, while preserving the object graph soundness.

1.4 Using Abstraction by Hierarchy in Object Graphs

For anything but the smallest systems, flat object graphs become very large, and as result, do not convey design intent. One way to make a large graph manageable is to collapse some nodes under other nodes. One way to do so is to use graph manipulation or transformation [26]. Another way to collapse one object under another is to create object hierarchy, where one object is the child of another. Instead of letting an object have child objects directly, we introduce an extra level of indirection, a *domain*, which is a named group of objects. So one object has one or more domains and each domain has one or more objects. Two types of domains express two forms of design intent: 1. strict encapsulation; or 2. logical containment.

Strict encapsulation (owned-by). An object `o1` *dominates* object `o2` if all paths from roots in the heap (typically a distinguished object and static fields) to `o2` go through `o1` [15]. In this scenario, `o2` is strictly encapsulated in the abstraction represented by `o1`, and so we show `o2` as owned-by `o1`, i.e., `o2` is in the private domain `owned` of `o1`. For example, the object `propMap` of type `HashMap` is strictly encapsulated in the object of type `BoardDrawing` (Fig. 1.1a). We depict a private domain with a thick, dashed border. Many ownership type systems enforce this notion of ownership, also called owner-as-dominator. Moreover, such a property can be inferred fully automatically. An object is either encapsulated or it is not.

Strict encapsulation, however, is too inflexible to make an object graph more hierarchical, after the fact, for code that was written without strict ownership in mind. When developers make an object owned by another, the object becomes inaccessible to other objects that still need to access it. As a result, if the developers do not change the code, they often have to leave the object at the same hierarchy level as other objects, i.e., more objects will be peers and the object graph is less hierarchical than intended. To make the graph more hierarchical,

developers need object hierarchy with fewer restrictions.

Logical containment (part-of). Another type of object hierarchy is *logical containment*. Sometimes, one object $o1$ is *conceptually* part-of another object $o2$, even if $o1$ is not owned-by $o2$. For example, `figureMap` of type `HashMap` is part-of the `BoardDrawing` object by being in its public domain (Fig. 1.1b). We depict a public domain with a thin, dashed border.

In general, it is hard to infer such a relationship from the code automatically, since it is by definition conceptual and reflects design intent rather than any code relationship. Object creation can often hint at logical containment, but not necessarily, as is the case with factory methods. We choose an underlying ownership type system that can express logical containment, Ownership Domains [11].

Using domain hierarchy to express design intent of object-oriented programs. When we say that object graphs express design intent, they have some of the following characteristics:

- They show each strictly encapsulated object inside a thick bordered domain inside its parent; such an object is not intended to be used, directly accessed, or mutated from outside of the object's boundary in order to maintain some important invariants, e.g., clients code cannot nullify the head of a list object (Fig. 1.1a);
- They show each logically contained object as grouped together with its parent; such an object is considered part of the object's public interface. For example, a `Node` object is part-of a `Graph` object, but it is intended to be accessed freely. The list of all `Node` objects needs to be encapsulated, however, to ensure the `Node` objects are properly disposed when the `Graph` object is disposed.
- They show top-level domains that express some architectural tiers such as Model and View;
- They do not show objects that are data structures or other implementation details at the top-level; we call these objects low-level objects;
- They show objects from the application domain at the top level; we call these objects

architecturally relevant;

- They promote both high-level understanding (when all top-level objects are collapsed, as in Fig. 1.4e) as well as details (when selected objects are expanded, as in Fig. 1.4d), due to the benefit of having a hierarchical representation, which allows expanding or collapsing object substructures
- They help show some structural design patterns compared to looking at the code. For example, when an object has a collection of other objects and the collection object points back to the parent object, it means there is the composite pattern, or when an object has a collection of objects that point to other objects, this could hint at an Observer pattern. An example of the composite design pattern can be seen in the JDepend subject system (Section 5.3), and in Fig. 5.2, the object of type `JavaPackage` points to a collection object of the same type, and the collection object points to the object of type `JavaPackage`. The object graph tools do not automatically highlight these patterns as their implementation may be subtle. The developers are also free to annotate the object graphs to highlight additional objects or edges of interest.

Thus, object graphs hold very relevant information to developers to improve their program comprehension, and make explicit several key facts that are implicit in the code. Therefore, object graph can directly help developers evolve the software in a way that maintains the invariants of the code, as confirmed by Ammar and Abi-Antoun [13]. This dissertation advances the state of the art for extracting and refining such object graphs. Our evaluation (Chapter 5) will show that the object graphs that our approach extracts from real code do in fact exhibit some of the above characteristics.

1.5 Ownership Type Qualifiers

One way to extract object graphs that convey design intent by hierarchy is to use ownership types. Using ownership types, developers can add additional information about object hierarchy in the code. There are different variations of ownership types. The key factor that differentiates various ownership type systems is the ownership context that they define,

and the rules about accessing objects in the ownership context. We list the three major variations of ownership types.

- **Ownership Types (OT):** Defines the ownership context to be an object [15]. Therefore, objects can be encapsulated inside other objects. OT implements the owner-as-dominator ownership model where an object is encapsulated or dominated by another object, i.e., all accesses to the dominated object must go through its dominator.
- **Universe Types (UT):** The ownership context is also an object, so an object can be in the ownership context of another object [18]. UT implements owner-as-modifier ownership model, which means the write access to the child object is restricted to its owner, but the other objects from outside of the context of the owner have read-only access to the child object without having to go through its dominator.
- **Ownership Domains (OD):** An ownership context is called *domain*, which is a named group of objects. Every object is a member of a single domain. There is a global domain with the reserved name `shared`. Moreover, each object can declare one or more domains to hold to its internal objects. A domain can be declared `public` or `private`. The `private` domains implement the owner-as-dominator ownership model. The `public` domains express part-of relation between objects. There is a reserved private domain for each object that is called `owned`. For any domain `d` declared on a class `C`, any two objects `n` and `n′` of the same class `C`, `n.d` and `n′.d` are distinct for distinct `n` and `n′`. There are domain links that define permissions to access between objects of different domains as explicit policy specifications. Having permission to access an object automatically grants access to its public domains. Objects that are inside the `owned` domain of an object can be accessed only by that object or by the objects that are in the sibling public domains. Other than the domain links, there are some implicit policy specifications such as an object has permission to access other objects in the same domain, or an object has permission to access to the objects in its declared domains. To make inference tractable, we simplify Ownership Domains into a

system we call Simple Ownership Domains (SOD) which we discuss later (Section 3.1).

1.6 Qualifiers vs. Object Graphs

To express object hierarchy, developers can add ownership type qualifiers directly to the code. However, adding qualifiers manually imposes a significant burden since each reference of a non-primitive type in the code needs a qualifier in order to type-check. Therefore, semi-automated or automated approaches for inferring these qualifiers are needed.

It is also hard to expect developers to directly understand the object structures from looking at code with qualifiers. Almost every single research paper on ownership types uses manually drawn object graphs to explain the object structure the qualifiers describe.

To illustrate how hard it is to understand what qualifiers to add, consider code (Fig. 1.2) with Ownership Domains qualifiers. For each object creation, different qualifiers are possible. At line 4, the object created at the object creation expression is in the same domain as the object of type `C1` (the declaring class). At line 5, the object is in the `owned` domain, and at line 6, the object is in the `PD` domain of the object of type `C1`. The object that is created at line 7 is in the domain `shared`, which is the global context. Each combination of actual domains produces a different object graph (Fig. 1.2). By showing developers object graphs extracted from the qualifiers and allowing them to edit the object graph directly, our WYSIWYG approach enables developers to choose qualifiers that express their design intent.

1.7 Key Idea: Interactive Refinement

In this section, we propose an approach, OOGRE (Ownership Object Graph Refinement Engine), for the interactive refinement of object graphs where developers do not directly change the qualifiers in the code.

To unclutter an object graph, developers can delete objects and their edges, but this makes the object graph unsound, since it no longer reflects all objects and their communication. Using OOGRE, developers use abstraction by hierarchy and move an object they no longer

```
class C1<owner, p> { // domain parameters
  private domain owned; // private domain
  public domain PD; // public domain
  obj = new C<owner, p>(); // Make peer of this
  obj = new C<owned, p>(); // Make owned-by this
  obj = new C<PD, p>(); // Make part-of this
  obj = new C<shared, p>(); // Place inside shared
}
```

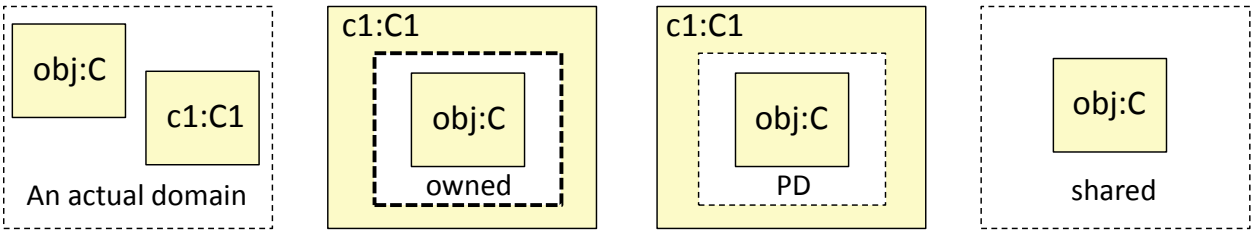


Figure 1.2: For the same code, different qualifiers are possible and produce different object graphs.

wish to see into a domain of another object. The object graph shows edges to the parent of the lowered object to account for any communication with the hidden object.

Developers interactively manipulate an object graph that is a sound abstraction of the runtime structure using a refinement, which changes the hierarchy relation between two objects. The following refinements are possible (Fig. 1.3):

- **MakeOwnedBy:** make an object *owned by* another by moving it into a private domain;
- **MakePartOf:** make an object conceptually *part of* another by moving it into a public domain;
- **MakePeerWith:** make an object peer with another object;
- **MakeShared:** make an object globally aliased by moving it into the domain `shared`;
- **SplitUp:** this composite refinement splits a merged object and moves the split object into another domain using one of four above types of refinements.

Our approach supports the above refinements, based on our experience manually adding qualifiers and refining object graphs on 100KLOC. These refined object graphs were used for conformance analysis, compared to as-designed architectural diagrams [5, 33], shown to

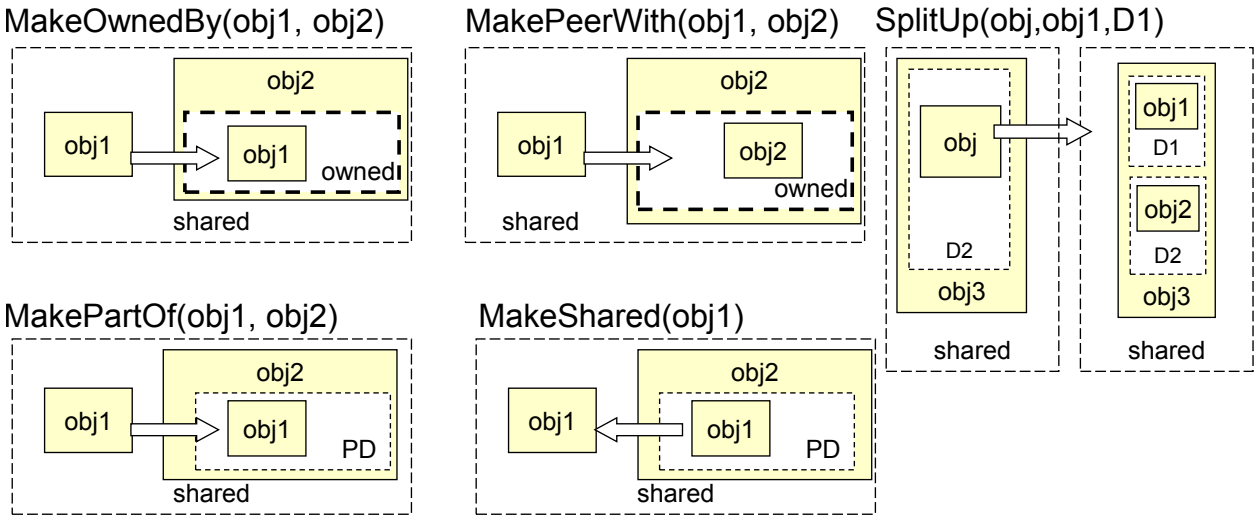


Figure 1.3: Five possible refinements, illustrated graphically.

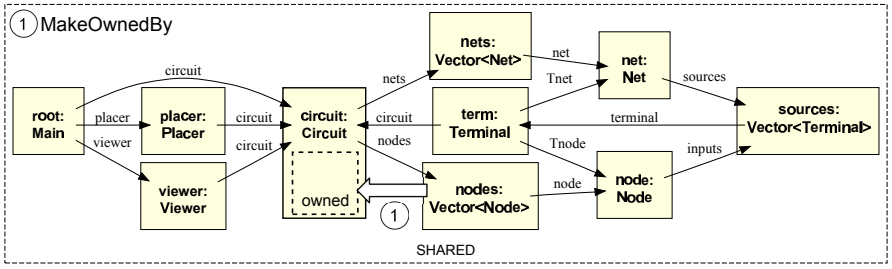
participants in a user study [13], and used to reason about security [7, 34].

Supporting the above refinements requires supporting certain qualifiers in the type system and defining a ranking among them. We discuss these modifiers later (Section 3.1) to place them in in the context of closely related work.

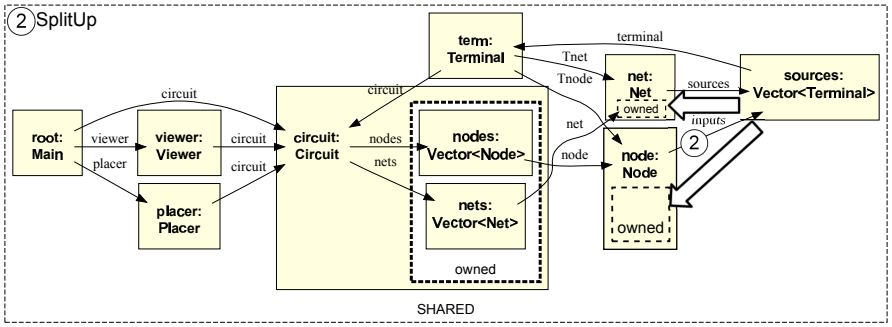
If the code supports the requested refinement, an inference analysis infers the corresponding qualifiers that type-check, and OOGRE extracts an updated object graph. We say this refinement is *done*. Otherwise, the refinement is *skipped*, the qualifiers in the code and the object graph are left unchanged. Based on the inferred qualifiers, an extraction analysis [5] extracts the refined object graph that the developers manipulate further. The ownership type system provides mathematical guarantees about the soundness of the inferred qualifiers and of the object graph. If the qualifiers type-check, the object graph is sound [5]. This dissertation reuses the extraction analysis by Vanciu [32] and Abi-Antoun [3], and contributes the inference analysis, as well as the integrated approach that combines extraction and inference.

1.8 Illustrative Example

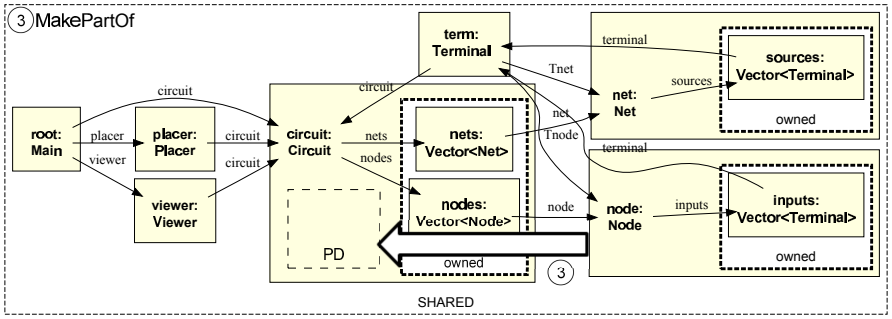
We illustrate the refinement of an object graph using MicroAphyds, a tiny example taken from a larger application, Aphyds [20]. In Fig. 1.4, the edges are points-to edges.



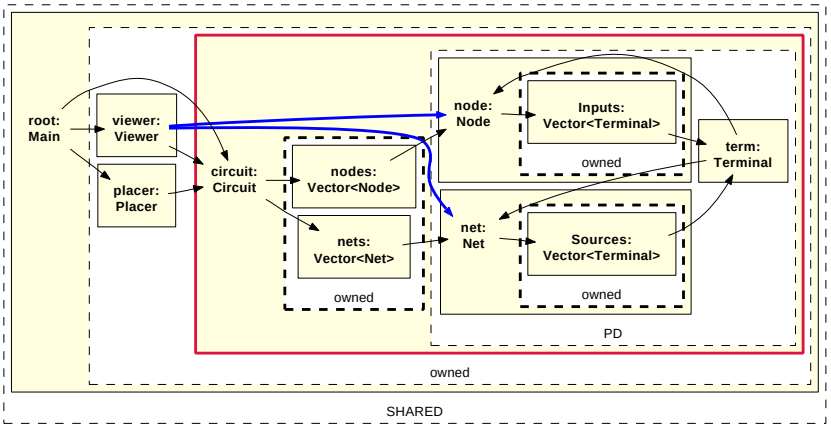
(a) Step 1: The flat graph has all objects in `shared`.



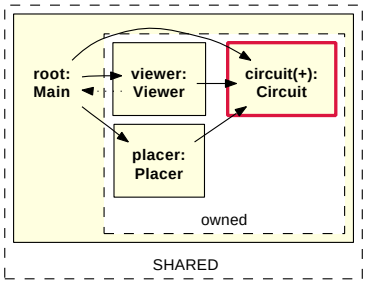
(b) Step 2: Move objects into `owned` of `circuit`.



(c) Step 3: Split objects to avoid excessive merging.



(d) Step 4: Pushing objects into `PD` of `circuit`.



(e) Step 5: Collapsing `circuit`.

Figure 1.4: MicroAphyds: refining a hierarchical object graph. Hierarchy enables collapsing several objects underneath the object of type `Circuit`.

Extracting a flat object graph. An initial rough object graph consists of a flat graph that is readily extracted without any developer input, by placing all the objects in the domain `shared` (Fig. 1.4a). While such a flat graph may be useful if it has a small number of objects, flat graphs can become too cluttered for larger systems. In a flat graph, it is hard to visually find an object or to follow the edges between different objects. In contrast, in a hierarchical graph, developers can collapse objects, and reduce the number of visible objects, as needed.

Another problem of placing objects in the same domain is that it leads the extraction analysis to excessively merge objects of the same type, which makes the graph less precise. For example, in the flat graph, one object of type `Vector<Terminal>` represents two object creation expressions in the code, which express conceptually different design intent.

Expressing strict encapsulation. Code quality tools such as FindBugs warn when an object returns an alias to a private field to other objects that may mutate it, a code quality issue called "representation exposure". Ownership type qualifiers can express and enforce this design intent and soundly avoid the representation exposure, even if occurs through intermediate assignments or method calls. In MicroAphyds, the developers note that the `Circuit` class has two `Vector` objects, and those objects should not be directly accessible to outside objects, which may mutate them and invalidate data structure invariants. As a result, they move the objects `nodes:Vector<Node>` and `nets:Vector<Net>` into the private domain `owned` of `circuit:Circuit` using two separate MakeOwnedBy refinements (Fig. 1.4b). If the code does not suffer from representation exposure, the refinement succeeds. If not, the refinement fails, with a message indicating the expression with the unexpected aliasing. For the refinement to be done, developers have to fix the code (i.e., remove the representation exposure by returning a copy) and re-attempt the refinement.

Splitting objects. The analysis that extracts the object graph merges runtime objects of the same type in the same domain into one object. If an object represents more than one object creation expression in the code, the developers are able to split the object by moving

```
class Circuit<owner, p> {
  private domain owned;
  public domain PD;
  Vector<Node><owned,p> nodes = new ...;
  Vector<Net><owned,p> nets = new ...;
  Node<p,p> node = new Node();
  Net<p,p> net = new Net();
  Terminal<p,p> term = new Terminal();

  void addNode(Node<p,p> nd) {
       this.nodes.add(nd);
  }
  void addNet(Net<p,p> net) {
       this.nets.add(net);
  }
}
```

(a) Qualifiers that correspond to Step 3.

```
class Circuit<owner, p> {                    1
  private domain owned;                      2
  public domain PD;                          3
  Vector<Node><owned,PD> nodes = new ...;    4
  Vector<Net><owned,PD> nets = new ...;      5
  Node<PD,p> node = new Node();              6
  Net<PD,p> net = new Net();                 7
  Terminal<PD,p> term = new Terminal();      8
                                             9
  void addNode(Node<PD,p> nd) {              10
       this.nodes.add(nd);                   11
  }                                          12
  void addNet(Net<PD,p> net) {               13
       this.nets.add(net);                   14
  }                                          15
}                                            16
```

(b) Qualifiers that correspond to Step 4.

Figure 1.5: The qualifiers behind the object graph at each step of refinement.

one of the objects that were merged in a different domain. In MicroAphyds, the developers split the object of type `Vector<Terminal>` into two objects of the same type and move one of them into the domain `owned` of `node:Node` as one refinement. Using another MakeOwnedBy, they move `sources:Vector<Terminal>` into `owned` of `net:Net` (Fig. 1.4c).

Expressing logical containment. Next, developers express that the objects of types `Node`, `Net` and `Terminal` are logically part of `circuit:Circuit`, so they move them into the public domain `PD` of `circuit:Circuit` (Fig. 1.4d).

The key idea is that logically contained objects are still accessible to the objects that have access to the parent. For example, the highlighted edges (appear as thick/blue) show that the `Viewer` object accesses the objects of type `Node` and `Net` that are part of `Circuit`, i.e., inside its public domain. With this hierarchy, developers can collapse the `Circuit` object, as can be seen in Fig 1.4e, thus hiding the objects it contains in its domains, and reducing the number of visible objects in the graph. The (+) on an object label indicates a collapsed object sub-structure.

1.9 Common Definitions

To clarify our contributions in this work, we reuse some of the terminology defined by Huang et al. [21]:

Actual Modifier: A member of the set of actual domains or actuals that can be used in SOD, namely `unique`, `lent`, `owned`, `n.PD`, `owner`, `p` and `shared`;

Qualifier: A *qualifier* is a pair of modifiers $<p, q>$;

Variable: A local variable, parameter, return, object creation, or field that requires a qualifier, i.e., that is not of a primitive type;

Maximal Qualifier: The highest ranked qualifier in the set of qualifiers of a variable that type-checks the variable's expressions;

Typing: Given a program P and an ownership type system F with a set of possible qualifiers Q_F, a typing T maps each variable in P to a qualifier in Q_F. A typing is *valid* for P in F if it renders P well-typed in F;

Maximal Typing: A typing that maps each variable to its maximal qualifier;

Set Mapping: In a set-based solution, a *Set Mapping* S maps each variable in program P to a set of possible qualifiers in type system F. Crucially, one set mapping may contain several valid typings and if one set becomes empty, the whole set mapping is discarded.

Optimality Property: The optimality property holds for a type system and a program if and only if the typing derived from the set-based solution by giving each variable the maximal qualifier from its set is a valid typing;

Conflict: When assigning the maximal qualifier for each variable does not type-check the program, the expression in the program that does not type-check is a conflict. Conflicts happen when the optimality property does not hold for a program and a type system.

1.10 Contributions

This dissertation contributes What You See Is What You Get (WYSIWYG) developer-driven inference of ownership type qualifiers. Developers preview an object graph, then manipulate or *refine* it to make it reflect their design intent. Behind the scene the inference analysis infers qualifiers that satisfy the requested refinement. Novel features of the inference analysis compared to closely related inference work include a larger set of qualifiers to support hierarchy with fewer restriction (logical containment) in addition to hierarchy with domination, as well as object uniqueness and object borrowing. The contributions of this work are as follows:

- **Enabling developers to refine an object graph directly to express their design intent by imposing hierarchy on objects.** We design a novel approach in which the developers refine an object graph by manipulating the object hierarchy. One type of refinement expresses strict encapsulation by moving an object into a private domain of another object. A second type of refinement expresses logical containment by moving an object into a public domain of another object. A third type of refinement makes two objects peers. A forth type of refinement move an object into the global domain. Finally, the last possible refinement is to split an object into two objects and move the split object into another domain using the four other types of refinements.
  Unlike the existing approach, where the developers add the qualifiers manually, extract the object graph, then go back to the code and modify qualifiers manually to refine the object graph, in the proposed approach, developers do not change qualifiers in the code manually. Instead, they only do refinements to express their design intent or to guide the analysis.
- **Utilizing a set-based solution to infer feasible SOD qualifiers including public domains.** To support the above approach, a whole-program static analysis utilizes a set-based solution to infer feasible qualifiers for a requested refinement. In the set-based solution, each variable maps to a set of qualifiers in a set mapping. After each

refinement, the analysis removes the infeasible qualifiers from the set of qualifiers of each variable. A refinement indicates the owning domain of the qualifier for some variables, and the analysis infers the ownership domain parameter, which can be any actual modifier in SOD including `n.PD`, in which `n` is the name of an object.

- **Utilizing a set-based solution to infer object borrowing and object uniqueness.** SOD supports object borrowing using the `lent` modifier, and object uniqueness using the `unique` modifier. We utilize the set-based solution to infer qualifiers containing those modifiers, by adding new adaptation functions and rank `unique` and `lent` as the highest ranked modifiers of SOD. Properties of `unique` and `lent` as a universal source and sink, respectively, make the set-based solution work for SOD efficiently.
- **Extract a valid typing that type-checks the program using the type rules of the SOD type system and satisfies the requested refinement, when the optimality property holds.** After computing a set mapping that contains feasible qualifiers, the analysis creates a typing using maximal qualifiers for each variable. If the optimality property does not hold, the maximal typing is not valid and does not type-check the program. The analysis handles the lack of the optimality property differently from how Huang et al. handle it for OT. They require developers to add qualifiers to help the analysis. Our approach provides two ways to resolve conflicts. First, the analysis warns developers about the conflict and developers do more refinements until the optimality property holds and the maximal typing type-checks the program. Second, developers run a special mode of the tool, the Assisted mode, which applies automated refinements on all the variables of the program that are have not been pinned down by a refinement to resolve conflicts.
- **Inferring qualifiers that express logical containment using public domains.** SOD express logical containment using public domains. In order to move an object into a public domain of another object, the analysis applies transfer functions based on the type of the expression. The transfer functions adapt the qualifiers containing

`PD` from the viewpoint of the receiver to be `n.PD`, using the adaptation functions that we define.

- **Defining a ranking over actuals of SOD, and extending it to rank qualifiers.** We define rankings over actual modifiers of SOD and extend it to ranked qualifiers. The criterion we use to define the rankings is to extract more hierarchical object graphs. Therefore, the analysis ranks higher the qualifiers that create more hierarchy in the object graph.
- **Supporting the Auto mode and Assisted modes to make the set-based solution converge on a typing.** To increase automation, the analysis applies automated refinements on key variables in the code instead of objects in the object graph. To find those variables, the analysis uses AST visitors. Using the automated refinements, Auto finds all the strictly encapsulated objects. If an object cannot be encapsulated, then Auto attempts to make them logically contained. If an object is neither strictly encapsulated nor logically contained, Auto attempts to make it peers with the other objects. The Assisted mode applied automated refinements to find a typing, when the developers are done applying refinements. It skips the variables that are pinned down by refinements that are done previously.

1.11 Thesis Statement and Hypotheses

Developers refine an object graph directly by making an object owned by another (strictly encapsulated), part of another (logically contained), peer of another, or globally aliased. If the code as-written supports the requested refinement, a static analysis infers valid qualifiers that type-check, otherwise the refinement is skipped. Using the above refinements, developers obtain hierarchical object graphs that apply abstraction by hierarchy and express their design intent.

We propose several hypotheses that are subordinate to the thesis statement. We list the success criteria of each hypothesis and support each hypothesis with evidence.

H1: Hierarchical Object Graph:

A hierarchical object graph leads to fewer visible objects at the higher levels of the object graph, since more objects are collapsed underneath other objects, compared to a flat object graph, in which all the objects are at the same hierarchy level.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- In a hierarchical object graph, it is possible to impose two types of hierarchy. The first is strict encapsulation, and the second is logical containment.
- Using SOD qualifiers, it is possible in practice to express an object hierarchy that provides architectural abstraction, where low-level objects such as data structures are at the lower levels of the hierarchy and more architecturally relevant objects from the application are at the higher levels;
- The static analysis to extract the object graph abstracts objects to pairs of types and domains. Developers can place objects of the same type in different domains and they will not get merged. Therefore, the hierarchical object graph is domain-sensitive, and developers assign properties to objects and write constraints on the structure of the graph [34];

Evidence. We support this hypothesis with the following evidence:

- For some subject systems, we use our approach to extract both a flat object graph and a hierarchical object graph. The analysis extracts a flat object graph by placing all the objects in the global domain `shared`. For each subject system, we use our approach to extract the hierarchical object graph by inferring SOD qualifiers. We compare the number of objects at the top level of a hierarchical object graph with the number of objects in the flat object graph.

H2: Logical Containment:

Applying both strict encapsulation and logical containment that are supported by SOD, compared to other type systems that support only strict encapsulation but no logical contain-

ment, leads to a more hierarchical object graph. A strictly encapsulated object is dominated by another object and all accesses to it must go through its dominator. A logically contained object is part-of another object, but still accessible to other objects.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- The developers express their design intent that is of the form of logical containment to group conceptually related objects and create more hierarchy;
- In a hierarchical object graph, objects that are at higher levels of the hierarchy access objects that are at the lower levels and are part of other objects;
- Without supporting logical containment in a hierarchical object graph, a lower level object stays a peer with its intended parent in order to remain accessible to other objects;
- In practice, real code exhibits the notion of logical containment to group related objects.

Evidence. We support this hypothesis with the following evidence:

- We use the analysis to extract hierarchical object graphs for small test cases as well as real world code bases by applying SOD that provides both strict encapsulation and logical containment. We also extract hierarchical object graphs using OT that provides only strict encapsulation. Then we compare the hierarchy in SOD and OT object graphs;
- We reuse some previously defined metrics [35], e.g., depth of hierarchy, which is the maximum depth of objects in a hierarchical object graph to compare different object graphs, and number of low-level objects in top-level domains;
- We compute the metrics on SOD and OT object graphs for each code base and compare them.

H3: Object Graphs vs. Qualifiers:

Using an interactive inference tool for SOD, developers refine an object graph to express their design intent by make owned by, make part of, make peer with and make shared re-

finements. Behind the scene, a static analysis infers valid qualifiers that type-check and the refinement is consider done. Otherwise the refinement is skipped and the qualifiers in the code and the graph are unchanged.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- Developers obtain a hierarchical object graph that expresses their design intent, by doing refinements and not modifying the qualifiers in the code;
- Developers perform refinements to express their design intent or to guide the analysis;
- The analysis builds on and preserves previous refinements. As a result, a refinement may be skipped if it contradicts a previous refinement;
- If a refinement is skipped, the analysis does not change the graph or the qualifiers resulting from previous refinements;

Evidence. We support this hypothesis with the following evidence:

- We implement the approach and run it on small (1 KLOC) and mid-size (5KLOC) subject systems;
- We run an independent type-checker after each refinement to validate the inferred qualifiers;
- We compare the inferred qualifiers with qualifiers that are inferred by another closely related work to compare the quality of the inferred qualifiers;
- We measure the cases where UT infers the qualifier `any`, but our analysis infers more precise qualifiers that convey more design intent;
- We study if using our approach, we refine an object graph to make it comparable to a diagram of the runtime structure that original developers of the system separately drew, and if it cannot be made comparable, we analyze the underlying reasons;
- For a skipped refinement, we investigate if the code does not support it, or if it can be done manually by adding qualifiers without using the tool, which we count as a false positive.

H4: Inferring SOD Qualifiers:

To express strict encapsulation and logical containment, the inference analysis constructs a set mapping that contains feasible qualifiers and multiple valid typings. Some of the feasible qualifiers contain ***`n.PD`****, where* ***`n`*** *is the name of an object, and* `PD` *is a public domain declared on the class of that object.*

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- The analysis infers qualifiers with `owned` as the owning domain for the strictly encapsulated objects using MakeOwnedBy refinements;
- All the objects that are strictly encapsulated in OT are also strictly encapsulated in SOD;
- If developers move a source object, o_1, into the `PD` domain of a destination object o_2 (using a MakePartOf refinement), and if other objects access o_1 from outside of the scope of o_2, for the variables in the code that those objects trace to, the analysis infers more precise qualifiers that contain `n.PD`, compared to `lent`. `n` in `n.PD` is the name of a `final` field or variable, or a sequence of `final` fields or variables in the code that traces to o_2. If there is no `final` final `n` in scope, the analysis infers `lent`.
- For each refinement, the analysis removes from the set mapping infeasible qualifiers that do not type-check, using its transfer functions. By infeasible qualifiers, we mean qualifiers that transfer functions remove from the set of qualifiers of a variable, since they do not type-check the expression;
- The analysis infers precise qualifiers, not trivial qualifiers that always type-check. For a visual approach, each object has to be in a precise domain, which is indicated by a precise qualifier. For qualifiers containing `lent` and `unique`, which are not precise, an extraction analysis resolves them to precise domains.

Evidence. We support this hypothesis with the following evidence:

- In practice, on real object-oriented code, developers can move an object into the `PD`

domain of another object that creates the source object;

- For each subject system, we measure the number of skipped refinements due to missing `final` fields or variables;
- We note the situations where the analysis tries to infer a precise qualifier, but the code does not support it, e.g., due to missing `final` modifiers.

H5: Finding a Valid Typing:

The inference analysis finds a typing from the set mapping using the maximal qualifiers for each variable. If the typing is not a valid typing, developers keep doing refinements until the analysis finds a maximal typing that type-checks the program. If the analysis cannot do so, developers start over with another sequence of refinements.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- In a set mapping that contains feasible qualifiers, there may be multiple valid typings that satisfy all the done refinements;
- The analysis first finds a typing by mapping each variable to its maximal qualifier. If the typing is a valid, i.e., type-checks the program, the analysis saves the typing as qualifiers in the code by saving the maximal qualifiers for each variable;
- If the typing is not valid, then the developers do more refinements until the analysis finds a valid typing using maximal qualifiers for each variable.
- In related approaches, the analysis may ask the developers to resolve cases where the analysis cannot find qualifiers that type-check, a situation that is called a conflict. Here, if the analysis cannot find a valid typing, developers perform additional refinements.

Evidence. We support this hypothesis with the following evidence:

- We measure after how many refinements the optimality property holds for the program and the SOD type system;
- For each subject system, we measure the number of refinements that make the analysis find a valid maximal typing and compare it to Huang et al. key metric that is qualifiers

per KLOC to find a valid maximal typing.

H6: Ranking Qualifiers:

The analysis ranks qualifiers that contain `unique` *and* `lent` *highest, followed by local domains (*`owned` *or* `PD`*), followed by domain parameters (*`owner` *or* `p`*). The lowest rank is the global domain (*`shared`*).*

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- The analysis ranks `unique` and `lent` as the highest ranked modifiers, because of their special properties. As a universal source a `unique` variable can be assigned to other variables. Other variables can be assigned to a `lent` variable, which is a universal sink;
- The inference analysis ranks local domains higher than domain parameters and ranks domain parameters higher that the global domains `shared`. The reason is local domains (`owned` or `PD`) create hierarchy, but domain parameters (`owner` or `p`) make objects peers. The defined ranking can be applied on qualifiers. Therefore, the qualifiers that contain local domains are ranked higher than qualifiers that contain domain parameters;
- The analysis ranks the actual `shared` as the lowest one, so it does not infer qualifiers that contain `shared` for any variable, if there are higher ranked feasible qualifiers. Therefore, the developers can move away from the trivial flat graph as much as possible. If the developers intentionally want to clearly separate a globally aliased object from other objects, the tool supports an explicit refinement, MakeShared for that purpose.
- A qualifier $<p, q>$ contains an owning domain p and an actual domain q. To rank two qualifiers, the analysis first ranks their owning domain then their domain parameter.

Evidence. We support this hypothesis with the following evidence:

- We compare the percentages of local domains in a valid typing, which is the result of the analysis, with a valid typing that is the result of applying OT to compare the hierarchy of the object graphs. Theses qualifiers approximate the shape or level of hierarchy in the object graphs;

- We compare the precision of the qualifiers of a valid typing with those of a typing that is the result of applying UT. Also, we compare the hierarchies of the resulting object graphs for SOD and UT.

H7: Automation:

The tool supports two additional modes for applying refinements, Auto, a fully automated mode when the developers do not do any manual refinements; and Assisted, once the developers finish applying refinements and they just want the tool to find a typing.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- In this automated mode, the analysis attempts three types of refinements: MakeOwnedBy to suggest strict encapsulation, MakePartOf to suggest logical containment, and MakePeerWith to suggest making objects peers;
  - Finding strictly encapsulated objects automatically is possible, since an object is either strictly encapsulated or not.
  - If a variable is a candidate for being strictly encapsulated is in fact not then it can be logically contained by the destination class;
  - If a candidate variable is neither strictly encapsulated nor logically contained, then it can be peer with the object of the destination class;
- The analysis uses AST visitors to find all the fields and local variables to apply automated refinements on them using the Auto and Assisted modes;
- The analysis does not apply automated refinements on the variables that are target variables of the refinements using the Assisted mode, in order to respect the previous refinements applied by the developers;

Evidence. We support this hypothesis with the following evidence:

- We measure the percentages of done auto-refinements of each type;
- We measure the portion of done auto-refinements out of the attempted ones;

1.12 Summary

The key idea in this dissertation is to enable developers to refine object graphs directly without having to change the qualifiers in the code. The qualifiers are hard to add to the code manually. We will discuss them in the next chapter. The proposed approach handles both the problem of:

- scaling the process of adding qualifiers to the code manually;
- letting the developer's refinements drive the inference, so the inferred qualifiers lead to an object graph that reflects the developer's design intent.

In most papers on ownership inference, researchers draw partial object graphs by hand. With this dissertation, the object graph is extracted using a tool and developers do not change the qualifiers in the code.

Outline. The rest of this dissertation is structured as follows. Chapter 2 gives some background on Ownership Domains and discusses the related approaches. Chapter 3 discusses the approach informally and positions this work in relation to related ownership type systems and inference of ownership type qualifiers. Chapter 4 formally describes the inference analysis. Chapter 5 evaluates the approach using two styles of evaluation, one comparative and one focusing on design intent. Finally, Chapter 6 discusses some implementation details, limitations, future work and concludes.

CHAPTER 2 BACKGROUND AND RELATED WORK

In this chapter, we first give some background on Ownership Domains and compare it with two other ownership type systems. Next, we introduces challenges that each ownership inference approach must handle. We then briefly discuss the most related approaches and introduce other related approaches, and how each approach handles the challenges.

2.1 Ownership Domains

In Ownership Domains [11], a class can declare one or more domains using the `domain` keyword (Fig. 1.2). Each instance of a class C gets a fresh instance of a domain d declared on the class; for distinct objects n_1 and n_2 of class C, the domains $n_1.d$ and $n_2.d$ are distinct.

In Ownership Domains, a class can take a number of *formal* domain parameters. Here, for simplicity, we allow just two, `owner` and `p`, e.g., `class C<owner, p> {...}`. A type is a class name and two *actual* domains, i.e., `C<p1,q1>`, where `p1` and `q1` are some domains or domain parameters in scope. Given an object that has a type `C<p1,q1>`, the first actual domain `p1` denotes the *owning domain* of the corresponding object. This is why we use the `owner` modifier for the name of the first domain parameter. When used as an actual owning domain on the type of an object o, `owner` means that o is in the same domain as the `this` object. The second actual domain `q1` denotes the *domain parameter* of the corresponding object. Ownership Domains can express the notion of temporary sharing of an object (object borrowing) using the `lent` modifier [12]. For example, an object that is passed to a method is used only for the duration of the call. An object can be borrowed from one domain to another as long as the second domain does not create a persistent reference to the borrowed object by storing it in a field. Only formal method parameters and local variables can be `lent`. Also, Ownership Domains expresses the notion of an object that has only one reference using

`unique` [12]. For example, an object that is created by a factory method does not belong to a specific domain and the client code determines its domain. Adding or inferring qualifiers involves adding a *pair* of actual domains `<p1,q1>` for a type, which we call *qualifier*.

2.2 Ownership Domains vs. Other Ownership Type Systems

There are similarities across the three ownership type systems covered in the closely related inference work [21] and this dissertation: Ownership Types (OT) [15], Universe Types (UT) [18], and Ownership Domains (OD) that we show in Table 2.1. In the rest of this dissertation, we refer to each system by its abbreviated name.

Similarly to OT and UT, OD can express the concept of an object that is strictly encapsulated by its outer context, and the concept of an object that has the same owning context. Similarly to OT, OD has the notion of an ownership parameter, and the concept of an object in the global context. Compared to OT and UT, OD has the notion of logical containment that is expressed using a public domain. Moreover, OD expresses object borrowing using `lent` and object uniqueness using `unique`.

In OT and UT, objects own other objects directly, i.e., the ownership context is an object. In OD, objects do not own other objects directly. Instead, a domain is an explicit, named, ownership context. Explicit contexts are important during the graphical refinement of an object graph. In our approach, developers drag an object and drop it into a named, explicit context. Otherwise, when developers move an object o_1 inside an object o_2, it would be unclear whether they are making o_1 owned-by o_2 or part-of o_2.

In contrast, UT can refer to an object o_1 in some other context but with a reference that cannot be used to mutate the referenced object (the modifier is `any`). As a result, UT requires additional purity qualifiers, which have to be either manually added or inferred using a separate inference analysis. Moreover, the modifier `any` does not provide any information about the actual ownership context of an object.

Table 2.1: Comparison across three ownership type systems. We show the corresponding modifier in the type system or "n/a" if the concept is not available in the type system.

	UT	OT	OD
global owner	n/a	`norep/world`	`shared`
strict encapsulation	`rep`	`rep`	`owned`
logical containment	n/a	n/a	`n.PD`
same owner as `this`	`peer`	`own`	`owner`
ownership parameter	n/a	`p`	`p`
temporary access	n/a	n/a	`lent`
only one reference	n/a	n/a	`unique`
readonly + pure	`any`	n/a	n/a
preference/ranking [1]	`any> rep > peer` [21, p. 9]	`rep > own > p` [21, p. 9]	`unique > lent > owned...` `... > n.PD > owner > p > shared`

2.3 Challenges of Inferring Ownership Qualifiers

Any approach to infer ownership qualifiers must address the following challenges.

- *Soundness:* Sound qualifiers implement a type system. A sound approach must infer qualifiers that type-check [12, 22, 19, 25, 31, 21, 17];
- *Precision:* An approach must select the most precise qualifier between valid qualifiers. The precision can be defined based one a preferred ranking over the qualifiers [22, 21] or the depth of the inferred ownership structure [25, 31, 37, 17].
- *Trivial qualifiers:* An approach must have a way of selecting a trivial qualifier that always type-checks and does not require expensive computation. The trivial qualifiers can be considered as a starting point for an approach, especially the ones that show the results of inference in graphical forms [25, 31, 37].
- *Interactive vs. fully-automated:* An approach can work in a fully-automated mode [12, 25, 19, 37] or in an interactive mode [22, 21, 17, 31]. An interactive approach may accept partial qualifiers and infer the remaining, or accept graphical interactions.
- *No solution:* An approach must handle the case when it cannot find any solution that type-checks. An approach may not save qualifiers [21, 17], or may produce meaningful error messages [31].

[1] We discuss the ranking of OD modifiers in Section 3.2

- *Multiple solutions:* An approach may infer more than one valid solution for a program, and it must be able to pick one. An approach may use metrics to pick between different solutions [37]. Another approach may show the different solutions to the developers and ask them to pick one [31].
- *Lack of information* The results of an interactive approach depend on the input qualifier from the developers. Sometimes, the provided qualifier from the developers is not enough for the approach to infer all the other qualifiers. In that situation, the ranking may not work. To resolve this, an approach may ask for information from the developers. For example, it may ask the developers to manually annotate all the object creation sites in the code [22] or a subset thereof [21].
- *Reusable code:* Parameters are often introduced to make code more reusable. Although it is hard to infer where the code is intended to be reusable automatically. Some approaches do not infer parameters, but at the cost of restricting the ownership model [19, 25]. Inferring an arbitrary number of parameters is often problematic [12]. For simplicity, an approach, including this one, may infer one parameter, which is still suitable to express a number of programs in practice [22, 21].

2.4 Closely Related Work

Our inference analysis instantiates the framework of Huang et al. [21] by providing a set of qualifiers, adaptation functions, and type-system-specific constraints. Our inference analysis computes a set-based solution, by starting with sets containing all possible qualifiers and iteratively removing ones that are inconsistent with the type system rules. We discuss in more detail the differences after we present the details of our approach (Section 3.9).

2.5 Other Related Work

We organize related approaches based on their output and their program analysis.

2.5.1 Static Analysis/Saves Qualifiers

Huang and Milanova [22] present an approach to infer OT qualifiers. An interactive approach that requires developers to add qualifiers for a subset of variables. The approach utilizes a set-based solution and uses transfer functions to analyze all the expressions and eliminate invalid qualifiers. The approach infers one ownership parameter and terminates with an error when there is no solution.

Vakilian et al. [31] propose a universal framework that accepts a type system and produces an inference for it. However, it requires a checker on top of the Checker framework [16] for the type system. The inference is interactive and inspired by speculative analysis to help developers decide the next steps, by showing the consequences of their decisions ahead of time. It builds a tree consisting of two kinds of node: error and change nodes.

Dietl et al. [17] build a tunable static inference for Generic UT. It works on Java programs with full, partial, or no qualifiers. By traversing AST, the approach generates constraints for variables and solves them by reusing a max-SAT solver, which limits the approach's scalability. The approach utilizes more than one strategy for multiple solutions: adjusting heuristics by changing the weights, or requiring developers to input partial qualifiers.

Aldrich et al. [12] present a type system called AliasJava and an algorithm to infer its qualifiers. AliasJava is similar to OD, but it does not support public domains. To infer alias parameters for each class, the algorithm conducts a constraint system including three sets of constraints, *equality*, *component*, and *instantiation* that guarantee soundness. The algorithm solves the constraints and integrates the result with other qualifiers based on a ranking. However, over 50% of the inferred qualifiers are `shared`. Moreover, the approach infers many alias parameters up to one for each field of one class.

Dymnikov et al. [19] present an ownership inference that infers `owned` qualifiers for the fields of a class. The inference implements some heuristics to infer strictly encapsulated fields in a class, so it is not a sound approach.

2.5.2 Static Analysis/Visualizes Ownership

Milanova and Vitek [25] present a static analysis that infers an ownership tree that follows the owner-as-dominator ownership model. First, it creates points-to sets using a points-to analysis. Second, an object graph is created using transfer functions that create edges to indicate the ownership relation between objects. Next, a dominance boundary analysis creates boundaries as subgraphs of the object graph.

Zhu and Liu [37] present a sound and fully-automated constraint-based ownership inference, Cypress that uses an application of linear programming. Cypress generates a visualized hierarchical decomposition of the heap statically. The hierarchy is based on ownership relations between the objects. Cypress follows the "tall and skinny" principle and favors heap decompositions that are taller and skinnier.

2.5.3 Dynamic Analysis/Saves Qualifiers

Dietl and Müller [36] present an approach that analyzes the execution of programs and infers ownership qualifiers from the executions. First, the approach builds the representation of object store that is called Extended Object Graph, which contains all the objects that ever existed in the store and their modification information. Next, it creates the dominator tree, since in UT, all the modifications of an object should be initiated by its owner. Then it resolves the conflicts with UT and harmonizes different instantiations of a class and outputs the qualifiers. The approach is fully-automated, but it is unsound since it uses dynamic analysis.

2.5.4 Dynamic Analysis/Visualizes Ownership

Rayside and Mendel [28] present an object ownership profiler to infer object ownership. They visualize the result of ownership inference in an *interactive* hierarchical matrix visualizer. They define the notion of Ownership Claim Graph (OCG). Therefore, they define the object x is the owner of the object y, if x is the immediate dominator of y in the program's OCG. They create a graph that shows which objects write on the other objects, and the

dominating objects can the owners of the modified objects.

2.6 Summary

In this chapter, by comparing Ownership Domains and two other ownership type systems, we notice that Ownership Domains is able to express logical containment ownership structure that is not expressible by other systems. Also, by identifying challenges of an ownership inference approach, we claim that OOGRE is the only interactive ownership inference approach where developers refine an object graph without changing the ownership qualifiers in the code. Moreover, OOGRE is the only approach that enables developers to express hierarchy with fewer restrictions using public domains. In the next chapter, we explain OOGRE informally and in the following chapter, we describe it formally.

CHAPTER 3 SET-BASED SOLUTION

In this chapter, we explain the initial set mapping and the initial object graph. Next, we discuss applying a refinement and running transfer functions. Then we explain how the inference analysis finds a maximal typing. Next, we discuss how the approach resolves conflicts. In the remainder of this chapter, when we say the analysis for brevity, we mean the inference, rather than the extraction analysis.

3.1 Simple Ownership Domains (SOD) Qualifiers

In order to make inference tractable, we simplify OD as follows, and call it Simple Ownership Domains (SOD). In SOD, we support the following: 1. a single private domain per class, hard-coded to be `owned`; 2. a single public domain per class, hard-coded to be `PD`; 3. an implicit domain parameter, `owner`, which is made explicit in the formalization and in the code examples; 4. a single explicit domain parameter per class, hard-coded to be `p`; and 5. implicit domain links [11] that make objects in private domains inaccessible to the outside, objects in public domains accessible, and objects in sibling domains accessible to each other. In the rest of this dissertation, we use SOD.

3.2 Ranking of Qualifiers

We define a ranking between the qualifiers that OOGRE may infer. The criterion for the ranking is to make the object graph more hierarchical, as is common in ownership inference [21]. First, we define a ranking between all the actuals of SOD (see the last row of Table 2.1). The actual `unique` is the highest ranked since it is the universal source and can be assigned any other modifier. The modifier `unique` may create hierarchy, if the extraction analysis resolves it to `owned` or `PD`. The actual `lent` does not create hierarchy, but `lent` is ranked the next highest since it allows variables with any modifiers to be assigned to it (see Section 3.9). After `unique` and `lent`, `owned` is the next highest ranked actual, since it creates hierarchy, and an `owned` object is strictly encapsulated. A public domain `PD` can

be referred to like a field, using the `n.PD` modifier (we discuss the syntax in Fig. 4.1). The next ranked actual is `n.PD`, which also creates hierarchy, but is less restrictive. Every object in `owned` can be in `PD`, but the reverse does not hold. In `n.PD`, `n` can be `this`. The fifth ranked is `owner`, which is a domain parameter and makes objects peers. The next ranked is `p`, which is also a domain parameter, and it may bind to any domain. The lowest ranked is `shared`, which gives no hierarchy and places objects in the global context.

We extend the ranking of modifiers to qualifiers. To determine the ranking of a qualifier, the analysis first considers the owning domain (p), then the domain parameter (q) of a qualifier ($<p, q>$). Therefore, to compare two qualifiers, the one that is higher ranked has a higher ranked owning domain. If two qualifiers have the same owning domain, then the one that has a higher ranked domain parameter is ranked higher.

3.3 Starting From an Initial Set Mapping

As the first step, the analysis maps each variable to an *initial set of qualifiers* that contains all the possible qualifiers by considering the kind of variable and the SOD constraints. In SOD, there are seven modifiers: `unique`, `lent`, `owned`, `n.PD`, `owner`, `p` and `shared`. The modifiers `unique` and `lent` can appear only as the owning domain in a qualifier, so there are 10 possible qualifiers that contain `unique` or `lent`. The modifier `owned`, internally qualified by the analysis as `this.owned`, can be the owning domain of 5 more possible qualifiers. For `n.PD`, `owner`, `p` as the owning domain, there are 4 more possible qualifiers, since they do not accept `owned` as a domain parameter. The `owned` domain is the private domain of an object, and using it as the domain parameter is incorrect, since it is inaccessible to the objects that are not in the peer `PD` domain. The modifier `shared` as the owning domain allows only the qualifier <`shared`, `shared`>, since a qualifier with `shared` as the owning domain should not have the domain parameter of a higher ranked modifier. Therefore, the full set of qualifiers contains 28 qualifiers, but all variables do not map to the full set. Moreover, we can use the type of the variable to dictate the initial set, following the notion of manifest ownership, and use this feature for `String`'s.

A field cannot be `lent`. We also do not support `unique` fields which require destructive reads. So a field variable maps to an initial set that contains 18 qualifiers. We do not support `unique` method parameters in order to infer more general method signatures. Method returns cannot map to qualifiers containing `lent`. So the initial set of qualifiers of method parameters and returns contain 23 members. Local variables map to the full set of qualifiers. The transfer functions invoke an intra-procedural standard Live Variables Analysis (LVA) to determine if `unique` is allowed. For simplicity, we omit the calls to the LVA.

For a variable in the `Main` class, the initial set of qualifiers is smaller, because `Main` does not declare the domain parameter `p`. Moreover, the owning domain of the root object is `shared`, so the domain parameter `owner` binds to `shared`, and there is no need to have qualifiers containing `owner`. We treat `owned` in `Main` as a public domain, so the qualifier <`this.PD`, `this.owned`> is possible, since we use the two top-level domains on the root object to express a two-tiered design. The root object is in `shared`.

Default qualifiers. After initializing the set mapping, the analysis saves for each variable default qualifiers that are guaranteed to type-check, as annotations in the code. A type-checker can then type-check them, and the extraction analysis can extract object graph. For the variables in the `Main` class, the analysis assigns the default qualifier <`shared`, `shared`>. The analysis assigns <`owner`, `owner`> to the variables in other classes. Based on the default qualifiers, the extraction analysis extracts an initial graph, which is flat, since all the objects in the initial object graph are in `shared`.

3.4 Applying Refinements

A refinement changes the set of qualifiers of one or more variables in the set mapping into a singleton set. Developers refine the object graph directly by drag-and-drop of a source object into a domain of a destination object. Then the analysis finds the set of variables that the source object traces to and changes their set of qualifiers in the set mapping accordingly. Those variable are the *target variables* of the refinement. We generalize MakeOwnedBy, MakePartOf, MakePeerWith and MakeShared into MakeX.

Each refinement operates on an OGraph G, and has a source OObject O_{src} and a destination OObject O_{dst}. The detailed representation of an OObject is used only by the extraction analysis [5] and is not needed here. The analysis translates the requested refinement in terms of variables, changes the set of qualifiers for the target variables in S, and runs the transfer functions to infer the qualifiers of the other variables.

MakeX. The MakeX refinement pushes a source object O_{src} into X as a domain or domain parameter of a destination object O_{dst} (Fig. 3.1). The analysis finds the target variables that O_{src} traces to, $\overline{x}$. Then it creates five instances of S, $S_{\texttt{owned}}$, $S_{\texttt{PD}}$, $S_{\texttt{owner}}$, $S_{\texttt{p}}$ and $S_{\texttt{shared}}$. The set of qualifiers of each target variable is modified to be a singleton in which the owning domain is X and the domain parameter is the same as the subscript of the corresponding S. For example, in $S_{\texttt{PD}}$, for a MakeOwnedBy, the set of qualifiers of a target variable is {<this.owned,this.PD>} and for a MakePartOf, it is {<this.PD,this.PD>}. If a target variable is in the Main class, the analysis creates only three instances of S, $S_{\texttt{owned}}$, $S_{\texttt{PD}}$ and $S_{\texttt{shared}}$. There is no domain parameter p in Main, and owner of the root object is shared, so no need to create $S_{\texttt{p}}$ and $S_{\texttt{owner}}$. Using the auxiliary judgement $mdbody()$, the analysis accesses the body of a method declaration (see Section 4.6). The analysis runs the transfer functions (highlighted in the rule) on each created S_q and all the expressions of the program to validate the changes and infer the other changes.

$$\frac{\begin{array}{c} O_{src} \in G \quad O_{dst} \in G \quad \overline{x} = getVars(O_{src}) \quad \forall x_i \in \overline{x} \quad Q_i = S[x_i] \\ \forall q \in \{\texttt{this.owned}, \texttt{this.PD}, \texttt{owner}, \texttt{p}\}\ \exists\ S_q\ s.t.\ Q'_i = \{<X, q>\} \in S_q \\ (x_i \rightarrow Q'_i)S_q \qquad \forall C \in CT, md \in C, e \in md, \quad \Gamma; S_q; n_{this} \vdash e, S'_q \end{array}}{S \xrightarrow{MakeX(G, O_{src}, O_{dst}, X)} S'_x}\text{[R-MX]}$$

$$\frac{\begin{array}{c} Q_{x_{src}} = S[x_{src}] \quad X = getDomain(O_{dst}, D_{dst}) \\ \forall q \in \{\texttt{this.owned}, \texttt{this.PD}, \texttt{owner}, \texttt{p}\}\ \exists\ S_q\ s.t.\ Q'_{x_{src}} = \{<X, q>\} \in S_q \\ (x_{src} \rightarrow Q'_{x_{src}})S_q \qquad \forall C \in CT, md \in C, e \in md, \quad \Gamma; S_q; n_{this} \vdash e,\ S'_q \end{array}}{S \xrightarrow{SplitUp(G, x_{src}, O_{dst}, D_{dst})} S'_x}\text{[R-SPU]}$$

Figure 3.1: MakeX: for MakeOwnedBy, X=this.owned, for MakePartOf, X=this.PD, for MakePeerWith, X=owner, and for MakeShared, X=shared. SplitUp splits and pushes the source variable x_{src} to the X domain of the destination object (X=this.owned, X=this.PD, X=owner or X=shared).

SplitUp. In the extracted object graph, objects of the same type and in the same domain get merged into one object. In a flat object graph where all the objects are in `shared`, an object may merge many object creation expressions in the code. Developers may want to split the object into distinct objects in different domains. To do so, they go to the traceability information associated with the object, select one object creation expression and move the corresponding variable into a domain of another object using MakeOwnedBy, MakePartOf, MakePeerWith, or MakeShared. Since the only difference between SplitUp and the other refinements is the number of target variables, we do not show SplitUp as an inference rule.

Respecting the whole set mapping. The analysis applies each refinement on a set mapping that is the result of the previous done refinements. This way, a current refinement cannot undo the result of a previous refinement. This leads to having smaller sets of qualifiers after each refinement, so the analysis can find a valid maximal typing sooner, since there are more sets that are singletons, so the set mapping is closer to become a typing. The drawback of respecting the whole set mapping is a previous refinement may lead the current refinement to get skipped by removing some qualifiers that were infeasible for that previous refinement, but not for the current one. Therefore, the developers may not be able to apply some refinements later. To avoid having a refinement that is skipped due to another one, one could respect only the set of qualifiers of the target variables of the previous done refinements, and after each refinement map all the other variables to their initial set of qualifiers. Respecting only the target variables makes the set mapping as flexible as possible for the next refinement. Although the latter seems to resolve the issue, it would discard all the work to remove the infeasible qualifiers from the sets of qualifiers. For each refinement, there would be a handful of target variables for which the sets of qualifiers would need to be respected. But the analysis would have to redo all the work. As result, the analysis would be less likely to find a typing.

3.5 Running Transfer Functions

The analysis applies transfer functions on each set mapping S, which is created by a refinement, and each expression in the program. Each transfer function takes S, an expression and produces an output S'. A transfer function removes infeasible qualifiers from the set of qualifiers of each variable (see Section 4.5). The transfer functions run until a fixed point when the sets of qualifiers of all variables no longer change. At the fixed point, for each variable x, S contains a set of qualifiers $S[x]$, which can be an empty set. If there exists a variable x for which $S[x]$ is empty, the entire S is discarded and not used any further. If after running the transfer functions, all the instances of S are discarded, then the refinement is skipped, the analysis does not save any qualifiers and the object graph is unchanged.

Multiple solutions. After running transfer functions, there may be more than one valid solution for the refinement. Each S represents a solution, so the analysis must select one to continue. To select S, the analysis uses a strategy that prefers S_q where q is `p`, `PD`, `owner`, `owned` and `shared` in this order. The analysis prefers a more flexible solution over the others. As a domain parameter, `p` can bind to any domain, so that is the most flexible solution. The modifier `PD` is more flexible than `owner` and `owned`, since an object that is in the public domain can be accessed by other objects. The modifier `owned` is the least flexible one, since an object in `owned` can be accessed only by the object that declares the domain, or by objects in its sibling `PD` domain.

Side effects of a refinement. Using the set-based solution, one refinement leads to some other objects moving into different domains in the refined object graph. Those changes are not due to the refinement, but they are in the inferred typing. We call those changes *auto-refinements*. There are auto-refinements, because in the set-based solution a variable maps to a set of qualifiers and using the ranking of the qualifiers, one qualifier is picked. In our current implementation, we respect the auto-refinements, because we preserve the set mapping after each refinement. So by doing the next refinement, developers still see the

changes due to the previous auto-refinements. We respect auto-refinements to avoid showing developers object graphs that differ dramatically after each refinement.

3.6 Finding a Typing that Type-checks from a Set Mapping

At this point, each variable in the current S maps to a set of qualifiers that are from multiple valid typings. Using the defined ranking between qualifiers, the analysis extracts a typing T from S by applying the max function on the set of qualifiers of each variable x in S. The max function picks the highest ranked, i.e., maximal, qualifier in the set of qualifiers of x. The typing T is the maximal typing.

$$T[x] = f(S[x]) \text{ where } f = max \text{ for all } x$$

Due to the nature of the set-based solution, extracting a valid typing is not simple. If the maximal typing T does not type-check the program, then the optimality property does not hold for the program and SOD. A key finding in Huang et al. is that for certain ownership type systems, one can derive a *unique maximal*, i.e., best, typing T from the set-based solution S. "The optimality property holds for a type system F and a program P if and only if the typing derived from the set-based solution S by typing each variable with the maximally/preferred qualifier from its set, is a valid typing."

For programs that have not been analyzed, and for the type systems such as SOD and OT, the optimality property does not hold and the analysis struggles to find a typing. To help the analysis to resolve conflicts, developers supply additional information as qualifiers strategically placed on certain variables. With these qualifiers, the optimality property may still not hold for type systems such as SOD and OT. The optimality property holds for programs with arbitrary qualifiers (including none) for UT due to some nice properties.

First, UT has the qualifier `any`, which type-checks many variables (due to subtyping of qualifiers), including fields, and is also the highest ranked. Second, unlike OT and SOD, UT does not have one ownership parameter per class, so a qualifier consists of a single modifier p, which leads to finding a maximal typing efficiently, unlike SOD and OT where the analysis must infer qualifiers that are pairs of modifiers $<p, q>$ and rank them.

Our key insight to make the set-based solution also work efficiently for SOD is adding `lent` and `unique`, which act as a universal source and sink, respectively (as discussed in Section 3.9). So `lent` and `unique` variables are assigned to variables that are mapped to more than one qualifier (see Section 4.2), which gives the transfer functions more choices of valid qualifiers for the left-hand side or the right-hand side of an assignment.

$$<\texttt{lent}, \_> \in S[x] \land <\texttt{lent}, \_> = max(S[x]) \land <\texttt{lent}, \_> = T[x] \implies$$
$$\forall <p, q> = T[y] \in S[y],\ T[x],\ T[y] \text{ type-checks } x = y$$
$$<\texttt{unique}, \_> \in S[y] \land <\texttt{unique}, \_> = max(S[y]) \land <\texttt{unique}, \_> = T[y] \implies$$
$$\forall <p, q> = T[x] \in S[x],\ T[y],\ T[x] \text{ type-checks } x = y$$

Moreover, the `unique` and `lent` modifiers are highly ranked, and by having them as the owning domain of qualifiers in the sets of the set mapping, the analysis picks those qualifiers using the *max* function to find a valid maximal typing very efficiently.

To make the optimality property hold, developers provide the analysis with additional information. In Huang et al., this information consists of partial qualifiers that developers add manually and that the analysis uses to initialize the sets with singletons and respect the qualifiers. In our approach, this information consists of refinements to express design intent and those to resolve conflicts and the analysis infers the qualifiers and overwrites existing ones in the code. By adding a qualifier manually, the developer provides both parts of the qualifier (p and q), but a refinement sets only the owning domain (p) and lets OOGRE infer

the actual for the domain parameter (q). We think asking developers to provide both p and q is harder, but fully evaluating this claim requires a user study. This paper focuses on the technical feasibility of the OOGRE.

In order to check the validity of T the analysis has to type-check it. The maximal typing T is a valid typing if, for each variable x, $T[x]$ type-checks the program. If T is a valid typing, the analysis saves its qualifiers to the code. If the maximal typing T does not type-check the program, it means the optimality property does not hold and there is at least one conflict to resolve. "A statement s is a conflict if it does *not* type check with the *maximal qualifier* derived from the set-based solution."

3.7 Resolving Conflicts

To resolve conflicts, Huang et al. require that developers guide the inference analysis by adding qualifiers to the code manually. In our approach, we propose three modes for resolving conflicts: Manual, Assisted or automatic (Auto).

Manual mode. After doing refinements to express their design intent, the developers may need to do more refinements to resolve conflicts until the optimality property holds. They continue doing refinements of their choosing. Or they let OOGRE guide them. OOGRE provides a MoreInfoNeeded window that indicates the conflict expression, its variables and the set of qualifiers they are mapped to. OOGRE also suggests a list of refinements to resolve the conflict. It is then the developers responsibility to propose a refinement that may resolve the conflict, by exploring the variables of the conflict expression and the set of qualifiers they map to. The developers resolve any remaining conflicts as they arise until the optimality property holds and OOGRE finds a typing.

Assisted mode. Developers may keep getting conflicts and may not be able to resolve them using more refinements. If the developers do not want to or cannot do those additional refinements, they launch the Assisted mode to have the analysis try more refinements on the variables (as opposed to refinements in terms of abstract objects). The Assisted mode uses

visitors to find remaining fields and local variables, beyond the target variables of the previous manual refinements that are done, so the previous refinements are respected. To resolve the remaining conflicts, Assisted attempts refinements. Assisted applies MakeOwnedBy on each field or local variable to find all the strictly encapsulated objects. Then it applies MakePartOf on each field or local that cannot be encapsulated. After all, if an object is not owned by some other object, it may be part of it. Finally, Assisted applies MakePeerWith on each remaining field or local variable that can be neither in `owned` nor in `PD`.

Auto mode. Developers analyzing an unfamiliar system or who do not wish to apply refinements use the Auto mode. In this push-button mode, OOGRE attempts the above automated refinements. Once Auto runs, developers cannot do any manual refinements, since Auto exhaustively considers all the variables. Unlike Assisted, Auto considers all fields and local variables and attempts the same automated refinement as above. The Auto mode also helps us test the tool. If all the automated refinements are skipped, it is likely that the transfer or adaptation functions are not correctly handling some language features.

Invalid sequence of refinements. If some of the manual or automated refinements contradict each other, the set mapping is discarded, i.e., one or more variables map to an empty set. It is possible to construct, using Manual, Auto or Assisted, a sequence of refinements that leads to a set mapping that is discarded. With no valid set mapping, OOGRE will not find a typing. A different set of refinements, however, may lead to a valid set mapping, and OOGRE may find a typing. If OOGRE cannot find a typing using the Assisted mode, the developers start over and apply another set of refinements. We show an example of an invalid sequence of refinements in Section 3.8.

Developer Interaction. The procedure to use OOGRE is as follows: 1. Developers launch OOGRE; 2. They specify the mode: Auto or Manual; 3. Using Auto, they push a button and wait for the results; 4. If the object graph does not express their design intent or if OOGRE cannot find a typing, they start over using the Manual mode; 5. Using the Manual mode, they continue performing refinements to express their design intent; 6. To resolve

```
class Main {
  Listener pieChart = new PieChart();
  Listener barChart = new BarChart();
  Listener model = new Model();
  void run() {
    model.addListener(barChart);
    model.addListener(pieChart);
    barChart.addListener(model);
    pieChart.addListener(model);
    model.notifyObservers();
    barChart.notifyObservers();
    pieChart.notifyObservers();
  }
}
```

Figure 3.2: Parts of the code from Listeners example.

conflicts, they optionally inspect a MoreInfoNeeded window about the conflict and perform a refinement; 7. They continue doing refinements until OOGRE finds a typing. 8. Or if they do not know how to resolve a conflict, they optionally launch the Assisted mode; 9. If OOGRE finds a typing, it saves the qualifiers of the inferred typing in the code and extracts an updated object graph; 10. If OOGRE cannot find a typing, developers start over and guide the analysis more carefully using different refinements.

3.8 Invalid Sequence of Refinements

We explain how an invalid sequence of refinements may prevent OOGRE from finding a typing, using an example. We show parts of the code from a small example called Listeners in Fig. 3.2. In this example, the classes `Model`, `BarChart` and `PieChart` are subclasses of an abstract class called `Listener`. So they override the methods `addListener` and `notifyObservers` from `Listener`. Based on the SOD type system rules, the qualifiers of an overridden method and all of its overriding methods must be the same.

Consider the following scenario that may happen using Assisted mode:

1. The developers launches OOGRE; 2. They apply a refinement to move the object of type `PieChart` inside the `PD` domain of the root object. The refinement succeeds and changes the set of qualifiers of the field `pieChart`;

$$MPO(\texttt{PieChart}, \texttt{Main}) \Longrightarrow S[\texttt{pieChart}] = \{<\texttt{PD}, \texttt{owned}>\}$$

3. After that refinement, the developers launch Assisted mode; 4. The analysis finds the set of target variables that contains variables such as `barChart` and `model`, but `pieChart` is not in the set, since it is the target variable of a manual refinement; 5. When the analysis starts applying the automated refinements in the Assisted mode, let's say it applies the following refinement to move `barChart` to the `owned` domain of the root object. This automated refinement succeeds and changes the set of qualifiers of the field `barChart`;

$$MOB(\texttt{barChart}, \texttt{Main}) \Longrightarrow S[\texttt{barChart}] = \{<\texttt{owned}, \texttt{PD}>\}$$

6. The analysis applies an automated MakeOwnedBy on `model` to move it into the `owned` domain of the root object, but the refinement is skipped, because at lines 6 and 7, the method `addListener` is called on two actual arguments `barChart` and `pieChart` with different qualifiers;

$$MOB(\texttt{model}, \texttt{Main}) \Longrightarrow S[\texttt{model}] = \emptyset$$

7. If the analysis attempts MakePartOf on `model` to move it to the `PD` domain of the root object, the refinement is skipped for the same reason;

$$MPO(\texttt{model}, \texttt{Main}) \Longrightarrow S[\texttt{model}] = \emptyset$$

8. The analysis never finds a typing in this scenario, because the set-based solution is over-constrained now with the different qualifiers for `barChart` and `pieChart`; 9. To solve this problem, the developers have to ensure that `barChart` and `pieChart` have the same qualifiers e.g., by applying the same manual refinements on them.

The problem can arise using either the Auto or the Manual mode. Using the Auto mode, the same scenario prevents OOGRE from finding a typing. If, similarly to the above, developers use the Manual mode and apply refinements on `barChart` and `pieChart` that lead to different qualifiers for them, OOGRE cannot find a typing.

3.9 Additional Contributions over Huang et al.

We extend the Huang et al. framework in important ways:

• Our graphical refinement approach requires showing each object in a domain in the object graph. As a result, the object graph extraction requires a typing with precise qualifiers, rather than a typing that uses general qualifiers such as `any` in UT;

• To support logical containment (MakePartOf refinements) and the SOD type system, we integrate public domains (`n.PD`) into the set-based solution, which requires new adaptation cases. Also, we impose fewer restrictions on the elements of a qualifier, e.g., `<p,owner>` is a valid qualifier in SOD whereas OT prohibits the parameter from being of a higher rank. Therefore, there are many more possible typings that can be extracted from one set mapping and we have bigger initial sets of possible qualifiers;

• In SOD, an actual domain can be `n.PD` where `n` is the name of an object. `n` can be `this` or a `final` field (or a sequence of `final` fields). The `n.PD` actual is the result of adapting `this.PD`, which is accessible from outside. In contrast, in OT, `n` is always `this` and there is no `PD`. The actual `this.owned` is not accessible from the outside, so it need not be adapted. For SOD, adaptation has to consider the different cases for `n`;

• To make the set-based solution work efficiently, we also include object borrowing (`lent`) and uniqueness (`unique`) [12], which also require new adaptation cases.

In ownership type systems, the assignment rule requires the left-hand and right-hand sides of an assignment to have compatible qualifiers. One type system, UT, adds flexibility by allowing qualifier subtyping, where `any` is the most general qualifier. Having a general qualifier that is highly ranked enables extracting a valid typing from a set mapping efficiently.

To gain both expressiveness, and to make the set-based solution work for SOD, we also allow more than strict equality at assignments using `lent` and `unique`. As a universal sink, a `lent` variable allows variables with of any other modifier to be assigned to it. As a universal source, a `unique` variable can be assigned to other variables of any other modifier. So having qualifiers that contain `lent` and `unique` in the set of qualifiers of some variables leads to keeping more qualifiers in the set of qualifiers of other variables in the set mapping. Furthermore, since `unique` and `lent` are highly ranked, the analysis finds a valid typing from the set mapping efficiently (as we discuss in Section 3.6). Indeed, a previous iteration of the analysis did not support `lent` and `unique`, and more often than not, it did not find a valid typing from a set mapping, since the sets of qualifiers for SOD are larger compared to OT or UT. Still, `lent` and `unique` bring their own challenge: they are highly ranked, and less precise than the other SOD modifiers. This challenge is addressed at the level of the extraction analysis, which uses a separate value flow analysis to resolve `lent` and `unique` into precise domains. We support the `lent` and `unique` highly ranked modifiers, which are more restrictive than `any`, less precise than the core OD modifiers, but can still be resolved to actual precise domains using a separate value flow analysis.

3.10 Implementation Considerations

In this section, we discuss the implementation considerations of our inference analysis. We first explain the data flow analysis that our inference is based on, then we show the user interface of OOGRE. Next, we explain some implementation details such as how we handle generic types and library code. At the end of this section, we explain a work around for having only one ownership parameter.

3.10.1 Data Flow Analysis

We implemented the inference analysis on a dataflow analysis framework, Crystal [1], which handles building the Control Flow Graph, the Three-Address Code representation, and invoking the transfer functions we supply. The analysis saves qualifiers as annotations in the code, using language support for annotations. We have an independent type-checker that reads the annotations and type-checks them, and a separate extraction analysis that uses the annotations to extract the object graph. We manually run the independent type-checker to validate the inferred qualifiers.

3.10.2 User Interface Prototype

Below, we show a screenshot of a working prototype of the OOGRE Eclipse plugin (Fig. 3.3), on the same MicroAphyds example. This is the prototype that we propose and use in our evaluation. The user interface is not this thesis contribution since it has not been evaluated with users yet.

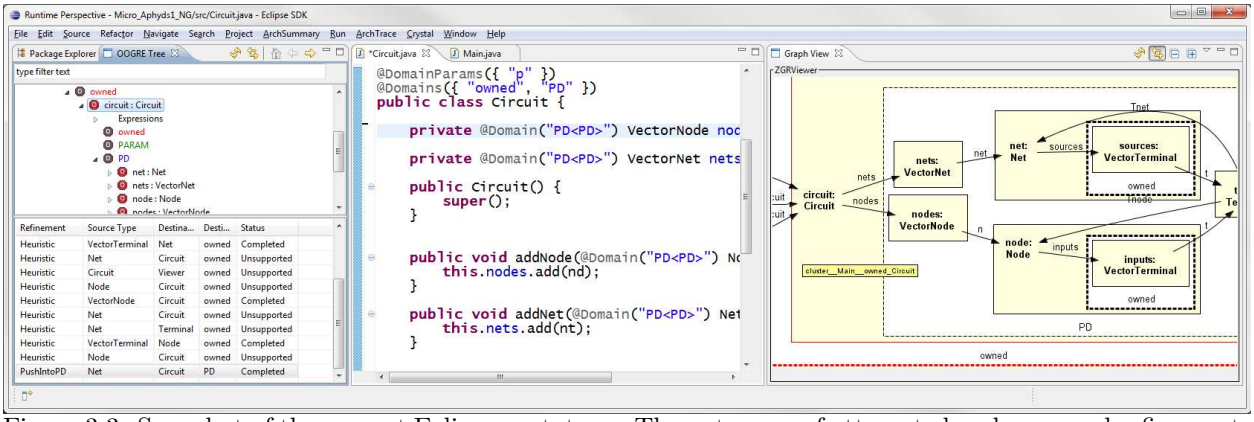


Figure 3.3: Snapshot of the current Eclipse prototype. The outcomes of attempted and proposed refinements (done or skipped) are shown below the ownership tree (bottom left).

The left side shows the ownership tree. Starting from the `SHARED` root domain, each object contains two domains, and each domain contains zero or more objects. By default, we create one private domain, called `owned`, and one public domain, called `PD`, per object.

Below the tree are the refinements that have been applied. The first column shows the refinement type, the middle columns show the arguments of the refinement, such as the

source object, the destination object, and the domain name. Finally, the last column shows the status of applying the refinement and running the inference analysis. If a refinement is done, it appears as Completed in the Status column. A skipped refinement has its status set to Unsupported. Refinements can be exported and re-applied to a system.

The middle part of the screen shows in the Eclipse Java editor the code with the annotations that are saved by the inference analysis. The annotations use language support for annotations available in Java 1.5 or later. The `@Domains` keyword lists the domains that a class declares. The `@DomainParams` keyword lists the domain parameters of a class. The `@Domain("p<q>")` annotation on a field, local variable, method parameter, or method return type, saves the inferred qualifier `<p,q>` for that variable. A separate type-checker can optionally check that the code and the annotations are consistent with each other and report any warnings in the Eclipse Problems window. The right side shows the refined graph based on running the graph extraction analysis on the code with annotations. The graph is visualized using nested boxes, which allow expanding or collapsing objects to reveal or hide lower-level objects.

```
class Vector<E><owner, p> {                                          1
          // E: generic type parameter                               2
              // owner, p: ownership parameters                      3
  E<p> obj; // the "trick" is to use one actual here                 4
  // obj is virtual/ghost field that summarizes Vector               5
                                                                     6
  // at usage point, p is replaced with two actuals                  7
  void addAll(Vector<E><lent,p> v) {                                 8
        ...                                                          9
  }                                                                  10
}                                                                    11
class Circuit<owner, p> { // domain parameters                       12
  private domain owned; // private domain                            13
  public domain PD; // public domain                                 14
  net = new Net<PD,p>();                                             15
  nets = new Vector<Net><owned, PD<p> >();                           16
  // 1. <PD,p> is qualifier of Net object                            17
  // 2. owned is actual for Vector's owner                           18
  // 3. PD<p> is actual for Vector's p                               19
  ...                                                                20
  nets.add( net ); // Add object to collection                       21
  net = nets.obj; // Field read                                      22
}                                                                    23
```

Figure 3.4: A generic collection with one generic type parameter requires a qualifier with an inner/nested modifier.

3.10.3 Generic Types

For expressiveness and to be able to run OOGRE on real code, we support generic types, e.g., a `Vector<E>` (Fig. 3.4), in which the type parameter `E` is replaced with a type at the declaration point. Without support for generics, OOGRE is not able to express the ownership information of the objects in a collection, leading it to skip refinements that should be done. Our inference analysis infers one additional "inner" parameter for a generic collection class with one generic type parameter. This is needed to express an object of type `Vector<E>` containing objects with the qualifier $<q, r>$. The `Vector` object is in some domain p and has an actual parameter $<q, r>$. Effectively, the qualifier of `Vector` is $<p, q<r>>$

In Fig. 3.5, we show one parametric rule, ADAPT-X-GEN. When $X = o$, it finds the result qualifier based on the inner and receiver qualifiers. When $X = i$, it finds the inner qualifier based on the result qualifier and the receiver qualifier. When $X = r$, it finds the receiver qualifier based on the result and the inner qualifiers.

$$\frac{\begin{array}{l}\textsc{Adapt-X-Gen}\\ t_1 = \texttt{<p>} \qquad t_2 = \texttt{<}p_0, \texttt{<}q_0, w_0\texttt{>>}\\ \quad n_{rcv} : \tau_{rcv} \qquad isGeneric(\tau_{rcv})\end{array}}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright_X t_1 = \texttt{<}q_0, w_0\texttt{>}}$$

Figure 3.5: Adaptation case to support generic types. X can be o for Adapt-Out, i for Adapt-In, and r for Adapt-Rcv.

We define new adaptation cases for one generic type parameter. In the rule that is shown in Fig. 3.5, the type variable qualifier, t_1, has only the owning domain. The reason is the receiver type is of a parameterized type, so it has to have an *inner* element and for the type argument the domain parameter and *inner* work like the owning domain and the domain parameter. Therefore, for the receiver qualifier, t_2, there are p_0 as the owning domain, q_0 as the domain parameter and w_0 as *inner*. In order to determine if a receiver type is a generic type, we use the auxiliary judgement $isGeneric(\tau_{rcv})$ that accepts the type of the receiver.

3.10.4 Library Code

Compared to Huang et al., we reason more carefully about library code. The current implementation handles library code in two ways. The first requires generating stubs for library classes. The inference analysis analyzes the stubs and infers qualifiers for them. As a result, we add support for generic types to analyze `java.util` collections (See Section 3.10.3). By using the ownership parameter, `lent`, and `unique`, we infer generally acceptable qualifiers for collections to enable their reuse across multiple applications [12]. The second way of analyzing libraries assumes that each library variable receives the default set of qualifiers {<`lent`, `p`>, <`owner`, `p`>, <`p`, `p`>}. The main advantage of stubs is being able to declare virtual fields (see Fig. 3.4). Without these, the object graph does not soundly reflect all the communication and is missing points-to edges between objects.

3.10.5 Working Around the Limitation of a Single Parameter

Ideally, an object of type `HashMap<K,V>` requires three domain parameters: one for the container itself, one for the key and one for the value. In SOD, with only two domain

parameters, we express a map by placing the key and value objects in the same domain and putting the container in another domain. This is problematic if we want either the key or the value object in `shared`, but not the other one, e.g., if the key is of type `String` and the value is an interesting architecturally relevant object.

To solve this problem, we specialize the `HashMap` into `StringKeyHashMap` where the type of the key is hard-coded to be a `String`. We then use manifest ownership (objects of a certain type are always in a certain domain) of `String` for the key object and save the domain parameter `p` for the value object.

We use the same idea to express a `HashMap` where its value object is also a container. We add a class called `MultiMap<K,V>`. The class `MultiMap` is a map, where the value object is of type `List<V>`. Ideally, the list object is not exposed to the outside, so it is owned by map. The method `get()` returns a fresh copy of the list, i.e., `unique`. The method `put()` accepts the key object and a corresponding list that is borrowed, so the qualifier for the list parameter of the method `put()` is `lent<p>`.

3.11 Additional Contributions Over Master's Thesis

This dissertation extends my Master's thesis [23] as follows:

- The system presented here is more scalable than the one in my Master's thesis. This system adds the option of resolving conflicts by applying refinements. Developers apply refinements to resolve conflicts, while in the previous version, the inference analysis resolves conflicts by enumerating all the possible combinations of qualifiers in the sets of qualifiers of variables. Enumerating all possible solutions makes the approach unscalable; if there are n variables in the program and each variable has m qualifiers in its set of qualifiers, there are m^n different combinations of qualifiers to consider for resolving a conflict. In this case, the function f from Section 3.6 is the $next()$ function, which considers all the possible qualifiers one by one. In this version of the inference

```
class StringKeyHashMap<V><owner,p> {
  // Key will be in 'shared<shared>'. Will not use 'p'
  String<shared,shared> key;
  V<p> value;
  ...
  V<p> get(String<shared,shared> key) {
       return this.value;
  }
  void put(String<shared,shared> key, V<p> val) {
       this.key = key;
       this.value = val;
  }
}
class MultiMap<K,V><owner,p> {
  private domain owned;
  K<p> key;
  List<V><owned,p> value;
  ...
  // The method get does not return an alias to the field this.value
  List<V><unique,p> get(K<p> key) {
    List<V><unique,p> newList = new ArrayList<V>();
    newList.addAll(this.value);
    return newList;
  }
  void putAll(K<p> key, List<V><lent,p> val) {
       this.key = key;
       this.value.addAll(val);
  }
}
```

Figure 3.6: Special classes for a map with String key and a map from an object to a list of objects.

analysis, when developers apply a refinement to resolve a conflict, the function f is the $max()$ function, which picks one qualifier. Therefore, to resolve a conflict, the analysis picks one qualifier for each variable (for the target variables, the picked qualifiers is imposed by the refinement). So, the time complexity to resolve conflicts is not exponential anymore (see Section 4.8).

- This dissertation adds more flexibility by introducing Auto and Assisted modes. The previous version of the analysis, OOGRE applies MakeOwnedBy and MakePartOf refinements on some variables in the code. However, using the Auto mode, OOGRE applies automated refinements (including MakePeerWith) on all fields and local variables. Using the Assisted mode, OOGRE applies the same refinements on fields and local variables beyond the target variables of the done manual refinements. In this

version of the inference analysis, developers resolve the remaining conflicts using the Auto and Assisted modes, and the time complexity is not exponential.

- This dissertation extends adaptation and transfer functions to infer object uniqueness (`unique`) and object borrowing (`lent`). It also incorporates the `unique` and `lent` modifiers into the defined ranking from Master's thesis, by adding them as the highest ranked modifiers, where `unique` is highest, followed by `lent`. The inference analysis also respects the type system specific constrains of the `unique` and `lent` modifiers.

3.12 Summary

In this chapter, we describe the analysis in detail informally. We discuss different states of the set mapping in the set-based solution such as initializing it, applying a refinements on it, running transfer functions on it, and finding a typing from the set mapping. We describe different modes of OOGRE and how and when to invoke them. We position our approach by highlighting its key differences with and extensions of closely related work. Finally, we discuss some implementation considerations and the additional contributions of this work over the Masters' thesis. The next chapter has a more formal description of the analysis.

CHAPTER 4 FORMALIZATION

In this chapter we formalize the inference analysis. We first discuss the abstract syntax that the analysis is based on. Then we show the adaptation cases for `owner`, `p` and `shared` modifiers along with the adaptation cases from SOD including `PD`, `unique` and `lent` cases. We formalize the set-level adaptation cases and transfer functions into inference rules, by respecting SOD constraints. Next, we formalize the class and method level rules, auxiliary judgements and SOD type-checking rules. Then we discuss the properties of the set-based solution and the statements of soundness and correctness of the inference analysis. Finally, we discuss the time complexity of different modes of the approach and compare it with the time complexity of the closely related work.

4.1 Abstract Syntax

We formalize our analysis by adapting Featherweight Domain Java (FDJ), which models a core of a Java-like language with Ownership Domains [11]. To enable comparisons with Huang et al., we simplify FDJ to the A-normal form and assume that each method has a single parameter, and each class has a single field. Of course, our implementation handles the general case. We also simplify FDJ to reflect the SOD simplifications such as hard-coded domain names and default domain links (See Section 3.1).

In our abstract syntax (Fig. 4.1), C ranges over class names; τ ranges over types; t ranges over qualifiers; f ranges over field names; v ranges over values; e ranges over expressions; x ranges over variable names; n ranges over values and variable names; the set of variables includes the distinguished variable `this` of type τ_{this} used to refer to the receiver of a method invocation, field read or field write; m ranges over method names; q and r range over formal domain parameters, actual domains, or the global domain `shared`; p ranges over `unique`, `lent` and all the other possible values for the actuals; an overbar denotes a sequence; the fixed class table CT maps classes to their definitions; a program is a tuple (CT, e) of a class table and an expression; Γ is the typing context; S defines a map from each variable to a set

of qualifiers; T defines a map from each variable to a single qualifier. $S[x]$ denotes reading the set of qualifiers for x in S; and $S' = [x \mapsto Q]S$ denotes updating the set of qualifiers for x in S.

$$
\begin{array}{rcl}
CT & ::= & \overline{cdef} \\
cdef & ::= & \texttt{class}\ C\texttt{<owner,p>}\ \texttt{extends}\ C'\texttt{<owner,p>}\ \{\ \texttt{domain owned;}\ \ dom;\ \ \tau\ f;\ md\ \} \\
dom & ::= & \texttt{public domain PD;} \\
md & ::= & \tau_R\ m(\tau\ x_m)\ \{\overline{\tau\ y}\ e; \texttt{return}\ y_m;\} \\
e & ::= & e;\ e\ \mid x = \texttt{new}\ C\texttt{<}p,q\texttt{>}()\ \mid y = x.f\ \mid y = \texttt{this}.f \\
 & & \mid x.f = y\ \mid \texttt{this}.f = y\ \mid x = y\ \mid x = y.m(z)\ \mid x = \texttt{this}.m(z) \\
n & ::= & x\ \mid\ v \\
q, r & ::= & \texttt{owner}\ \mid \texttt{p} \mid\ n\texttt{.PD}\ \mid\ \texttt{this.owned}\ \mid\ \texttt{shared} \\
p & ::= & \texttt{unique}\ \mid \texttt{lent} \mid\ q \\
\tau & ::= & C\texttt{<}p,q\texttt{>}\quad \textit{Type} \\
t & ::= & \texttt{<}p,q\texttt{>}\quad \textit{Qualifier} \\
x, y, z & \in & \textit{Variables} \\
\Gamma & ::= & x\ \rightarrow\ \tau\quad \textit{Static typing context} \\
S & ::= & \emptyset\ \mid\ S \cup \{x \mapsto \{\texttt{<}p,q\texttt{>}\}\}\quad \textit{Set Mapping (SM)} \\
T & ::= & \emptyset\ \mid\ T \cup \{x \mapsto \texttt{<}p,q\texttt{>}\}\quad \textit{Typing}
\end{array}
$$

Figure 4.1: Abstract syntax for SOD, adapted from Featherweight Domain Java (FDJ) [11].

Since in `n.PD`, `n` can be `this`, the adaptation has to distinguish between the inner `this` and the outer `this`. To avoid capture during adaptation, we substitute `that` for the inner `this` using [`that/this`]. In the transfer functions, after adaptation, `that` is substituted with `this` ([`this/that`]) for the inner `this`, and the outer `this` is substituted with the corresponding object name `n`, using [`n/this`], if there is `n.PD` in the resulting set.

Assumptions. The inference runs, after the fact, on an existing Java-like program that type-checks, and inserts the ownership type qualifiers. In a formalization of SOD, in contrast to a formalization of a Java-like language, a type $\tau = C\texttt{<}p,q\texttt{>}$ has two orthogonal components: the class name C and the ownership type qualifier ¡p,q¿. Similarly to Huang et al., we treat the ownership type system as orthogonal to or independent from the Java type system. As a result, the inference rules for the transfer functions (Fig. 4.5) do not include the Java sub-typing checks. Those are in the type-checking rules in Section4.6 (Fig. 4.6). Running our inference analysis on a Java-like program that does not type-check may lead to undefined inference results.

4.2 New Adaptation Cases

Adaptation cases for `owner`, `p` and `shared` are similar to the cases in Huang and Milanova [22] for Ownership Types. We show them in Section 4.3. We do not have any restriction over the values of owning domain and domain parameter of a qualifier, so we add an extra case for adaptation where the inner qualifier is <`p`, `owner`>.

The qualifier of an expression is the result of an adaptation, when the receiver of the expression is not `this`. In each adaptation case, there is an inner qualifier, a receiver qualifier and a result or outer qualifier. More formally, we say t_{out} is the result of adapting t_{in} from the viewpoint of t_{rcv}.

$$t_{rcv} \triangleright t_{in} = t_{out}$$

An inner qualifier can be the qualifier of a field, a method parameter, or a method return. Other than `owner`, `p` and `shared`, an inner qualifier can contain `this.PD`, `lent` or `unique`, so we need new adaptation cases to handle them. We show the general adaptation rule in Fig. 4.2. As the inner qualifier, t_1 may contain `this.PD`, `lent` or `unique`. For t_1, we substitute `this` with `that`, since t_1 is the inner qualifier, and the corresponding variable is not declared in the class of `this`. Later on, in the transfer functions, `that` is substituted with `this` again. The qualifier of the receiver, t_2, can be any qualifier, so we show it as $<p_0, q_0>$. The result of adaptation, t_3, is based on t_1 and t_2. Writing this in inference rule format leads to nearly identical rules. Instead, we use a tabular form to show t_1, t_3 and the name of the individual cases in Table 4.1.

For example, in the rule ADAPT-D-D, t_1 has `that.PD` as its owning domain and domain parameter. Therefore, independent from t_2, t_3 is <`n.PD`, `n.PD`>, and `n` is a final field of the same type as the receiver declared in the current class. In the rule ADAPT-D-O, the domain parameter of t_1 is `owner`, so in the result qualifier, the owning domain of t_2 is selected as the domain parameter, and t_3 is <`n.PD`, p_0>.

For the `PD` cases, if there is no `final` field `n` of the receiver type, instead of `n.PD`, the

Table 4.1: Adaptation cases for `PD`. X is `lent` or `unique`. t_1 is the inner qualifier and t_3 is the result qualifier. _ is for don't care, n/a is for an adaptation case that does not apply.

Rule name	t_1	t_3	
		`n` is final	`n` is not final
ADAPT-A-O	<p, owner>	$<q_0, p_0>$	$<q_0, p_0>$
ADAPT-D-D	<that.PD, that.PD>	<n.PD, n.PD>	n/a
ADAPT-D-O	<that.PD, owner>	<n.PD, p_0>	<lent, p_0>
ADAPT-D-A	<that.PD, p>	<n.PD, q_o>	<lent, q_o>
ADAPT-D-S	<that.PD, shared>	<n.PD, shared>	<lent, shared>
ADAPT-O-D	<owner, that.PD>	$<p_0$, n.PD>	n/a
ADAPT-A-D	<p, that.PD>	$<q_0$, n.PD>	n/a
ADAPT-S-D	<shared, that.PD>	<shared, n.PD>	n/a
ADAPT-X-D	<X, that.PD>	<_, n.PD>	n/a
ADAPT-X-O	<X, owner>	<_, p_0>	<_, p_0>
ADAPT-X-A	<X, p>	<_, q_o>	<_, q_o>
ADAPT-X-S	<X, shared>	<_, shared>	<_, shared>

$$\frac{\Gamma; n_{this} \vdash n_{this} : C_{this}<p_{this}, q_{this}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n \vdash t_2 \triangleright t_1 = t_3} \text{ ADAPT-GEN}$$

Figure 4.2: General rule for the adaptation of `n.PD`, `lent` and `unique`. We show a case analysis for t_1 and t_3 in Table 4.1. p_0 and q_0 can be any modifier.

analysis infers `lent`. For example, in the rule ADAPT-D-O, if the analysis cannot find a `final n` in scope, the result of adaptation is <lent, p_0>. Since `lent` cannot appear as the domain parameter of a qualifier, for the rule ADAPT-D-D, the result of adaptation cannot be <lent, lent>. In that case, the adaptation fails and the transfer functions remove the qualifier <this.PD, this.PD> from the set of qualifiers of inner variable.

We also show the adaptation cases for `lent` and `unique` in Table 4.1, which are similar. The modifiers `lent` and `unique` can occur only as the owning domain of a qualifier, so we have four cases for each one. When the owning domain of the inner qualifier is `lent` or `unique`, the owning domain of the outer qualifier can be any modifier or *don't care*, which we show by _ in the cases. Using *don't care*, the adaptation cases for `lent` and `unique` reflect how values can flow to `lent` and `unique` (see Section 3.9).

The judgement form for adaptation is as follows:

$$\Gamma; n_{this}; n_{rcv} \vdash <p_x,\ q_x> \triangleright <p_y,\ q_y> = <p_z,\ q_z>$$

4.3 Other Adaptation Cases

$$\textsc{Adapt-O-O}\quad \frac{t_1 = <\texttt{owner}, \texttt{owner}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <p_0, p_0>}$$

$$\textsc{Adapt-O-A}\quad \frac{t_1 = <\texttt{owner}, \texttt{p}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <p_0, q_0>}$$

$$\textsc{Adapt-A-A}\quad \frac{t_1 = <\texttt{p}, \texttt{p}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <q_0, q_0>}$$

$$\textsc{Adapt-O-S}\quad \frac{t_1 = <\texttt{owner}, \texttt{shared}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <p_0, \texttt{shared}>}$$

$$\textsc{Adapt-A-S}\quad \frac{t_1 = <\texttt{p}, \texttt{shared}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <q_0, \texttt{shared}>}$$

$$\textsc{Adapt-S-S}\quad \frac{t_1 = <\texttt{shared}, \texttt{shared}> \qquad t_2 = <p_0, q_0>}{\Gamma; n_{this}; n_{rcv} \vdash t_2 \triangleright t_1 = <\texttt{shared}, \texttt{shared}>}$$

Figure 4.3: Adaptation cases for `owner`, `p` and `shared`.

For completeness, in Fig. 4.3, we include adaptation cases for `owned`, `owner`, `p` and `shared`, which are similar to the cases in Huang and Milanova [22] for OT, and `rep`, `own` and `p` and `norep`, respectively.

4.4 Set-level Adaptation

In the set-based solution, in order to handle all the possible combinations of qualifiers, each transfer function uses three types of adaptation functions that operate on sets of qualifiers. First, Adapt-Out ($\triangleright_o$) adapts qualifiers of the outer variable by accepting qualifiers of the inner and the receiver (n_{rcv}) variables as input. Second, Adapt-In ($\triangleright_i$) adapts qualifiers of the inner variable by accepting qualifiers of the outer variable and the receiver variable as input. Third, Adapt-Rcv ($\triangleright_r$) adapts qualifiers of the receiver variable by accepting qualifiers of the outer variable and the inner variable as input.

The judgement form for set-level adaptation is as follows, where Q_i is a set of qualifiers:

$$\Gamma; n_{this}; n_{rcv} \vdash Q_1 \triangleright_X Q_2 = Q \text{ where } X = o \text{ or } X = i \text{ or } X = r$$

ADAPT-OUT

$$\frac{\forall t_{in} \in Q_{in}, \forall t_{rcv} \in Q_{rcv},\ \Gamma; n_{this}; n_{rcv} \vdash t_{rcv} \triangleright t_{in} = t_{out} \qquad t_{out} \in Q_{out}}{\Gamma; n_{this}; n_{rcv} \vdash Q_{in} \triangleright_o Q_{rcv} = Q_{out}}$$

ADAPT-IN

$$\frac{\forall t_{out} \in Q_{out}, \forall t_{rcv} \in Q_{rcv},\ \Gamma; n_{this}; n_{rcv} \vdash t_{rcv} \triangleright t_{in} = t_{out} \qquad t_{in} \in Q_{in}}{\Gamma; n_{this}; n_{rcv} \vdash Q_{out} \triangleright_i Q_{rcv} = Q_{in}}$$

ADAPT-RCV

$$\frac{\forall t_{out} \in Q_{out}, \forall t_{in} \in Q_{in},\ \Gamma; n_{this}; n_{rcv} \vdash t_{rcv} \triangleright t_{in} = t_{out} \qquad t_{rcv} \in Q_{rcv}}{\Gamma; n_{this}; n_{rcv} \vdash Q_{out} \triangleright_r Q_{in} = Q_{rcv}}$$

Figure 4.4: Set-level adaptation functions.

4.5 Transfer Functions

Our transfer functions generalize the transfer functions in Huang et al. to handle SOD qualifiers with `PD`, `lent` and `unique`. A transfer function accepts an expression and S and accesses the set of qualifiers of the variables of the expression in S. By intersecting the sets of qualifiers of variables, a transfer function removes the infeasible qualifiers from the set of qualifiers of the variables. Then it updates the sets of qualifiers of the corresponding variables in S and creates S'. Fig. 4.5 shows the inference rules for each transfer function. For the transfer functions that require adaptation, we handle qualifiers that contain `n` in `n.PD`. We highlight in gray our extensions to the Huang et al. rules. The judgement form for the transfer function over an expression e is as follows:

$$\Gamma; S; n_{this} \vdash e,\ S'$$

Object creation. The rule TF-NEW transfers over an object creation expression that consists of a left-hand side variable x and a call to the constructor of the class C. The qualifier $<p, q>$ and the class C form the type of the object being created. Due to SOD

TF-NEW

$$\frac{p \in \{\texttt{unique}, \texttt{owned}, \texttt{PD}, \texttt{owner}, \texttt{shared}\} \quad S' = [x \to (S[x] \cap \{<p, q>\})]S}{\Gamma; S; n_{this} \vdash x = \texttt{new}\ C<p, q>(),\ S'}$$

TF-ASSIGN

$$\frac{<\texttt{lent}, q_x> \notin S[x] \implies <\texttt{lent}, q_y> \notin S[y] \quad S' = [x \to (S[x] \cap S[y]), y \to (S[x] \cap S[y])]S}{\Gamma; S; n_{this} \vdash x = y,\ S'}$$

TF-FIELDRW

$$\frac{\begin{array}{c} <\texttt{lent}, q_f> \notin S[f] \quad <\texttt{unique}, q_f> \notin S[f] \\ \Gamma; n_{this}; x \vdash S[x] \triangleright_o [\texttt{that/this}]S[f] = Q_o \quad <\texttt{lent}, q_f> \notin S[f] \implies <\texttt{lent}, q_o> \notin Q_o \\ \forall t_o \in Q_o\ s.t.\ t_o = <x.\texttt{PD}, q> \text{ or } t_o = <p, x.\texttt{PD}> \text{ or } t_o = <x.\texttt{PD}, x.\texttt{PD}> \exists t \in S[y]\ s.t.\ [x/\texttt{this}]t = t_o \\ Q_o \cap (S[y] \leftarrow t_o) = Q_y \quad \Gamma; n_{this}; x \vdash Q_y \triangleright_i S[x] = Q_i \quad Q_i \cap S[f] = Q_f \\ \Gamma; n_{this}; x \vdash Q_y \triangleright_r Q_f = Q_r \quad Q_r \cap S[x] = Q_x \quad S' = [y \to Q_y,\ f \to [\texttt{this/that}]Q_f, x \to Q_x]S \end{array}}{\Gamma; S; n_{this} \vdash x.f = y \text{ or } y = x.f,\ S'}$$

TF-THISFIELDRW

$$\frac{<\texttt{lent}, q_f> \notin S[f] \quad <\texttt{unique}, q_f> \notin S[f] \quad S' = [y \to (S[y] \cap S[f]), f \to (S[y] \cap S[f])]S}{\Gamma; S; n_{this} \vdash \texttt{this}.f = y \text{ or } y = \texttt{this}.f,\ S'}$$

TF-INVK

$$\frac{\begin{array}{c} mdbody(m) = (x_m, y_m) \quad <\texttt{lent}, q_{y_m}> \notin S[y_m] \quad <\texttt{unique}, q_{x_m}> \notin S[x_m] \\ \Gamma; n_{this}; y \vdash S[y] \triangleright_o [\texttt{that/this}]S[x_m] = Q1_o \quad <\texttt{lent}, q_{x_m}> \notin S[x_m] \implies <\texttt{lent}, q1_o> \notin Q1_o \\ \forall t1_o \in Q1_o\ s.t.\ t1_o = <y.\texttt{PD}, q> \text{ or } t1_o = <p, y.\texttt{PD}> \text{ or } t1_o = <y.\texttt{PD}, y.\texttt{PD}> \exists t1 \in S[z]\ s.t.\ [y/\texttt{this}]t1 = t1_o \\ Q1_o \cap (S[z] \leftarrow t1_o) = Q_z \quad \Gamma; n_{this}; y \vdash Q_z \triangleright_i S[y] = Q1_i \quad Q1_i \cap S[x_m] = Q_{x_m} \\ \Gamma; n_{this}; y \vdash S[y] \triangleright_o [\texttt{that/this}]S[y_m] = Q2_o \quad <\texttt{lent}, q_{y_m}> \notin S[y_m] \implies <\texttt{lent}, q2_o> \notin Q2_o \\ \forall t2_o \in Q1_o\ s.t.\ t2_o = <y.\texttt{PD}, q> \text{ or } t1_o = <p, y.\texttt{PD}> \text{ or } t2_o = <y.\texttt{PD}, y.\texttt{PD}> \exists t2 \in S[x]\ s.t.\ [y/\texttt{this}]t2 = t2_o \\ Q2_o \cap (S[x] \leftarrow t2_o) = Q_x \quad \Gamma; n_{this}; y \vdash Q_x \triangleright_i S[y] = Q2_i \quad Q2_i \cap S[y_m] = Q_{y_m} \\ \Gamma; n_{this}; y \vdash Q_z \triangleright_r Q_{x_m} = Q1_r \quad \Gamma; n_{this}; y \vdash Q_x \triangleright_r Q_{y_m} = Q2_r \quad Q1_r \cap Q2_r \cap S[y] = Q_y \\ S' = [z \to Q_z, x_m \to [\texttt{this/that}]Q_{x_m}, x \to Q_x, y_m \to [\texttt{this/that}]Q_{y_m}, y \to Q_y]S \end{array}}{\Gamma; S; n_{this} \vdash x = y.m(z),\ S'}$$

TF-THISINVK

$$\frac{\begin{array}{c} mdbody(m) = (x_m, y_m) \quad <\texttt{lent}, q_{y_m}> \notin S[y_m] \quad <\texttt{unique}, q_{x_m}> \notin S[x_m] \\ S' = [z \to (S[z] \cap S[x_m]), x_m \to (S[z] \cap S[x_m]), x \to (S[x] \cap S[y_m]), y_m \to (S[x] \cap S[y_m])]S \end{array}}{\Gamma; S; n_{this} \vdash x = \texttt{this}.m(z),\ S'}$$

Figure 4.5: Transfer functions.

constraints, the object cannot be created in `lent` or `p`. The rule intersects the set of qualifiers of x with the qualifier $<p, q>$, which means the qualifier of x is $<p, q>$.

Assignment. The rule TF-ASSIGN extracts the set of qualifiers of the left-hand side (x) and right-hand side (y) variables and intersects them. If there is no qualifier with `lent` as the owning domain in the set of qualifiers of x, the set of qualifiers of y cannot contain `lent` neither. The reason is a `lent` variable cannot be assigned to a variable that is not `lent`. Finally, the rule updates the sets of qualifiers of both variables in S and creates S'.

Field read and write. We show one transfer function for field read and write expressions

(the rule TF-FieldRW). A field cannot map to qualifiers that contain `lent` or `unique`. First, the rule substitutes `this` with `that` for the qualifiers of the field f, since f is not declared in the class of `this`. To compute the updated set of qualifiers for y, Adapt-Out computes a set of qualifiers, Q_o. The set Q_o cannot contain `lent`, since the set of qualifiers of f does not contain `lent`. So the rule removes any qualifier with `lent` as the owning domain from Q_o. Moreover, for each qualifier t_o in Q_o in which t_o contains x.`PD` as the owning domain, the domain parameter, or both, if there is a qualifier t in set of qualifiers of y ($S[y]$) where instead of x.`PD`, t contains `this.PD`, the rule substitutes `this` with x for t. Then the rule intersects Q_o with $S[y]$, and the result is the new set of qualifiers for y, Q_y. To compute the new set of qualifiers of f, Adapt-In computes a set of qualifiers, Q_i using Q_y. Then, the rule intersects Q_i with $S[f]$ The result is Q_f, which is the new set of qualifiers for f. Next, for the receiver x, Adapt-Rcv computes a set of qualifiers using Q_y and Q_f, which is Q_r. The rule intersects Q_r with $S[x]$ to find the new set of qualifiers for x, Q_x. In Q_f, the rule substitutes `that` with `this` and updates the set of qualifiers of the variables in S to generate S'.

Field read and write with `this` as the receiver. When the receiver of a field read or a field write expression is `this`, there is no need for adaptation. Therefore, the rule TF-ThisFieldRW intersects the sets of qualifiers of the left-hand or the right-hand sides with the set of qualifiers of the field. The rule ensures that there is no qualifier that contains `lent` or `unique` in the set of qualifiers of the field. The rule creates S' by updating the set of qualifiers of the variables in S.

Method invocation. In the input expression of the rule TF-Invk, x is the left-hand side variable, and y is the receiver. The y_m variable represents the return of the method. The variable z is the argument of the method invocation. By calling the *mdbody*() auxiliary judgement (see Section 4.6), the rule extracts x_m that is the formal method parameter. The rule asserts that a method parameter cannot be `unique` as it is unsupported in this version, and a method return cannot be `lent`. First, the rule substitutes `this` with `that` in the sets

the qualifiers of x_m and y_m, since they are not declared in the class of `this`. The rule does ADAPT-OUT using the sets of qualifiers of y and x_m and the resulting set is $Q1_o$. The result of ADAPT-OUT may contain `lent`. A qualifier with `lent` as the owning domain in $Q1_o$ is valid, if the set of qualifiers of x_m contains `lent`. Otherwise, the rule removes `lent` from $Q1_o$. For each qualifier $t1_o$ in $Q1_o$, if $t1_o$ contains y.`PD` as the owning domain, the domain parameter, or both, if there is a qualifier $t1$ in set of qualifiers of z ($S[z]$) where instead of y.`PD`, $t1$ contains `this`.`PD`, the rule substitutes `this` with y for $t1$. The set $Q2_o$ is the result of ADAPT-OUT using the sets of qualifiers of y and y_m. Again, the rule validates `lent` qualifiers in $Q2_o$ and substitutes `this` with y for each qualifier $t2$ in $S[x]$, if there is a corresponding qualifier $t2_o$ in $Q2_o$ containing y.`PD`. By intersecting $Q1_o$ with $S[z]$ and $Q2_o$ with $S[x]$, the rule computes the new sets of qualifiers for z and x, which are Q_z and Q_x, respectively. By applying ADAPT-IN on Q_z and $S[y]$, the rule computes a set of qualifiers $Q1_i$ and the result of intersecting it with $S[x_m]$ is the new set of qualifiers for x_m, Q_{x_m}. Again, by applying ADAPT-IN on Q_x and $S[y]$, and intersecting the result with $S[y_m]$, the rule computes the new set of qualifiers of y_m, Q_{y_m}. For y, the rule does ADAPT-RCV and computes the result using Q_z, Q_f, Q_x and Q_m. Then it intersects the result of ADAPT-RCV with $S[y]$ and computes Q_y, the new set of qualifiers for y. For x_m and y_m, `that` is substituted with `this` in the computed sets of qualifiers.

Method invocation with `this` as the receiver. There is no need for adaptation, if the receiver of a method invocation is `this`. The rule TF-THISINVK ensures that the set of qualifiers of the method parameter does not contain `unique` and the set of qualifiers of the method return does not contain `lent`. The rule intersects the set of qualifiers of the formal method parameter with the actual method argument and updates the sets of the corresponding variables with the resulting set in S. Also, it intersects the set of qualifiers of the left-hand side with the method return variable and updates their sets of qualifiers in S and creates S'. The higher-level rules and the auxiliary judgements are shown in Section4.6.

$$\frac{\Gamma(x) = C_x\!<\!p_x, q_x\!> \qquad C <: C_x \qquad <\!p_x, q_x\!> = <\!p, q\!> \qquad p_x \in \{\texttt{owned},\ \texttt{PD},\ \texttt{owner},\ \texttt{shared}\}}{\Gamma \vdash x = \texttt{new}\ C\!<\!p, q\!>()}\ [\text{T-New}]$$

$$\frac{\Gamma(x) = C_x\!<\!p_x, q_x\!> \qquad \Gamma(y) = C_y\!<\!p_y, q_y\!> \qquad C_y <: C_x \qquad <\!p_x, q_x\!> = <\!p_y, q_y\!> \qquad p_x \neq \texttt{lent} \Longrightarrow p_y \neq \texttt{lent}}{\Gamma \vdash x = y}\ [\text{T-Assign}]$$

$$\frac{\Gamma(x) = C_x\!<\!p_x, q_x\!> \qquad \Gamma(y) = C_y\!<\!p_y, q_y\!> \qquad \tau_f\ f \in CT(C_x) \qquad \tau_f = C_f\!<\!p_f, q_f\!> \qquad p_f \neq \texttt{lent} \qquad p_f \neq \texttt{unique} \qquad C_y <: C_f \qquad <\!p_x, q_x\!> \triangleright <\!p_f, q_f\!> = <\!p_y, q_y\!> \qquad p_y \neq \texttt{lent} \qquad pubsig(f)}{\Gamma \vdash x.f = y}\ [\text{T-Write}]$$

$$\frac{\Gamma(y) = C_y\!<\!p_y, q_y\!> \qquad \tau_f\ f \in CT(C_{this}) \qquad \tau_f = C_f\!<\!p_f, q_f\!> \qquad p_f \neq \texttt{lent} \qquad p_f \neq \texttt{unique} \qquad C_y <: C_f \qquad <\!p_f, q_f\!> = <\!p_y, q_y\!> \qquad p_y \neq \texttt{lent}}{\Gamma \vdash \texttt{this}.f = y}\ [\text{T-ThisWrite}]$$

$$\frac{\Gamma(x) = C_x\!<\!p_x, q_x\!> \qquad \Gamma(y) = C_y\!<\!p_y, q_y\!> \qquad \tau_f\ f \in CT(C_y) \qquad \tau_f = C_f\!<\!p_f, q_f\!> \qquad p_f \neq \texttt{lent} \qquad p_f \neq \texttt{unique} \qquad C_f <: C_x \qquad <\!p_y, q_y\!> \triangleright <\!p_f, q_f\!> \ = \ <\!p_x, q_x\!> \qquad pubsig(f)}{\Gamma \vdash x = y.f}\ [\text{T-Read}]$$

$$\frac{\Gamma(x) = C_x\!<\!p_x, q_x\!> \qquad \tau_f\ f \in CT(C_{this}) \qquad \tau_f = C_f\!<\!p_f, q_f\!> \qquad p_f \neq \texttt{lent} \qquad p_f \neq \texttt{unique} \qquad C_f <: C_x \qquad <\!p_f, q_f\!> = <\!p_x, q_x\!>}{\Gamma \vdash x = \texttt{this}.f}\ [\text{T-ThisRead}]$$

$$\frac{mdtype(m) = \tau_m \to \tau_r \qquad \Gamma(x) = C_x\!<\!p_x, q_x\!> \qquad \Gamma(y) = C_y\!<\!p_y, q_y\!> \qquad \Gamma(z) = C_z\!<\!p_z, q_z\!> \qquad \tau_m = C_m\!<\!p_m, q_m\!> \qquad \tau_r = C_r\!<\!p_r, q_r\!> \qquad C_r <: C_x \qquad C_z <: C_m \qquad <\!p_y, q_y\!> \triangleright <\!p_m, q_m\!> = <\!p_z, q_z\!> \qquad p_m \neq \texttt{lent} \Longrightarrow p_z \neq \texttt{lent} \qquad <\!p_y, q_y\!> \triangleright <\!p_r, q_r\!> = <\!p_x, q_x\!> \qquad p_r \neq \texttt{lent} \qquad pubsig(m)}{\Gamma \vdash x = y.m(z)}\ [\text{T-Invk}]$$

$$\frac{mdtype(m) = \tau_m \to \tau_r \qquad \Gamma(x) = C_x\!<\!p_x, q_x\!> \qquad \Gamma(z) = C_z\!<\!p_z, q_z\!> \qquad \tau_m = C_m\!<\!p_m, q_m\!> \qquad \tau_r = C_r\!<\!p_r, q_r\!> \qquad C_r <: C_x \qquad C_z <: C_m \qquad <\!p_z, q_z\!> = <\!p_m, q_m\!> \qquad p_m \neq \texttt{lent} \Longrightarrow p_z \neq \texttt{lent} \qquad <\!p_r, q_r\!> = <\!p_x, q_x\!> \qquad p_r \neq \texttt{lent} \qquad pubsig(m)}{\Gamma \vdash x = \texttt{this}.m(z)}\ [\text{T-ThisInvk}]$$

Figure 4.6: Typing rules for SOD, adapted from OD [11] and including `lent` and `unique` [12]

4.6 SOD Typing Rules

It is important that the transfer functions only include qualifiers in the set mapping that type-check the corresponding expression. By applying the SOD typing rules, the transfer functions guarantee that. After finding a maximal typing, the SOD typing rules are used in order to reason about the correctness of the qualifiers in the typing. We adapt the typing rules

$$\frac{\tau\ f \qquad pubsig(C,f) \qquad md\ OK\ in\ C}{\texttt{class}\ C\texttt{<owner,p> extends}\ C'\texttt{<owner,p>}\ldots\ OK}[\textsc{ClsOK}]$$

$$\frac{\begin{array}{c}CT(C) = \texttt{class}\ C\texttt{<owner,p> extends}\ C'\texttt{<owner,p>}\ldots \\ override(m, C'\texttt{<owner,p>}, \tau \to \tau_R) \\ \tau = C\texttt{<}p_1,q_1\texttt{>} \quad \tau_R = C_r\texttt{<}p_2,q_2\texttt{>} \quad mdtype(m) = \tau' \to \tau'_R \quad \tau' = C'\texttt{<}p_3,q_3\texttt{>} \\ \tau'_R = C'_r\texttt{<}p_4,q_4\texttt{>} \quad pubsig(m) \quad \texttt{<}p_1,q_1\texttt{>} = \texttt{<}p_3,q_3\texttt{>} \quad \texttt{<}p_2,q_2\texttt{>} = \texttt{<}p_4,q_4\texttt{>}\end{array}}{\tau_R\ m(\tau\ x)\ \{\ \overline{\tau\ y}\ e;\texttt{return}\ y_m;\ \}\ OK\ in\ C}[\textsc{MethOK}]$$

Figure 4.7: SOD type system constraints.

for SOD to this framework (Fig. 4.6). We expand the rules from Ownership Domains [11] to include special cases for when the receiver is `this`, as for the transfer functions. Most crucially, we adapt the rules to use viewpoint adaptation instead of substitution of formals to actuals in FDJ [11]. In SOD, there is no subtyping between qualifiers, just qualifier equality. Also, SOD imposes its own type system constraints, such as prohibit object creation with an owner being `p` (T-New). An object can be created only in a local domain of `this` or its own domain.

Two rules work on a higher level than variables or expressions (Fig. 4.7). MethOK ensures that for an overriding method the qualifier of the method parameter is the same as the qualifier of the method parameter of the overridden method, and similarly, for the return type. Moreover, if method is a `public` method, then its parameter or its return type cannot have `owned` in their qualifiers. ClsOK checks that a `public` field cannot be `owned`, and that all the methods in a class are valid based on MethOK.

The auxiliary judgements *mdType*() and *mdBody*() return the type and the body of a method, respectively. The auxiliary judgement *pubsig*() enforces the SOD constraints on the qualifiers of `public` methods and fields. First, $pubsig(C, f)$ checks if the field f in the class C has the visibility modifier `public`, its qualifier cannot contain `owned`. Also, $pubsig(m)$ checks if a method is `public`, then the qualifier for its method parameter or its return cannot contain `owned` (Fig. 4.8).

$$\textsc{Aux-mdtype}\ \frac{(\tau_R\ m(\tau\ x)\ \{\overline{\tau\ y}\ e;\ \texttt{return}\ y_m;\})\ \in\ md}{mdtype(m) = \tau \to \tau_R} \qquad \textsc{Aux-mdbody}\ \frac{(\tau_R\ m(\tau\ x)\ \{\overline{\tau\ y}\ e;\texttt{return}\ y_m;\})\ \in\ md}{mdbody(m) = (x,\ y_m)}$$

$$\textsc{Aux-mpublic}\ \frac{\begin{array}{c} public(m) \qquad mdtype(m) = \tau \to \tau' \\ \tau = C{<}p, q{>} \qquad \tau' = C'{<}p', q'{>} \\ p \neq \texttt{owned} \qquad q \neq \texttt{owned} \\ p' \neq \texttt{owned} \qquad q' \neq \texttt{owned} \end{array}}{pubsig(m)} \qquad \textsc{Aux-fpublic}\ \frac{\begin{array}{c} \tau\ f \in CT(C) \qquad public(f) \\ \tau = C'{<}p, q{>} \\ p \neq \texttt{owned} \qquad q \neq \texttt{owned} \end{array}}{pubsig(C, f)}$$

Figure 4.8: Auxiliary judgements.

4.7 Properties of Set-Based Solution

Since our inference analysis instantiates the Huang et al. by providing the set of qualifiers, adaptation functions, and SOD constraints, unsurprisingly, the soundness property of the inferred qualifiers holds. We adapt their Proposition 1 and show it holds for our set-based solution. Proposition 1 states if the set-based solution removes a qualifier from the set of qualifiers of a variable, then there is no valid typing that contains the removed qualifier.

Proposition 1. *Let S be the set-based solution. Let x be any variable in a program P, and let $<p, q>$ be any qualifier in SOD. If $<p_0, q_0> \notin S[x_0]$ for some x_0, then there does not exist a valid typing T for program P in SOD such that $T[x] = <p_x, q_x>$ and $<p_x, q_x> \in S[x]$ for all x and $T[x_0] = <p_0, q_0>$.*

Proof. (Sketch) We say that $<p, q>$ is a *valid qualifier* for x if there exists a valid typing T, where $T[x] = <p, q>$. Let x_0 be the first variable that has a valid qualifier $<p_0, q_0>$ removed from its set $S[x_0]$ and let f_e be the transfer function that performs this removal. Since $<p_0, q_0>$ is a valid qualifier for variable x_0 in expression e, for the other variables in e, there exist other valid qualifiers $<p_1, q_1>$,...,$<p_n, q_n>$ that make e type-check in SOD. If $<p_1, q_1> \in S[x_1], ..., <p_n, q_n> \in S[x_n]$, then by definition of a correct transfer function, f_e would not have removed $<p_0, q_0>$ from $S[x_0]$. So one of $x_1, \ldots, x_n$ must have had a valid qualifier removed from its set *before* the application of f_e. This contradicts the assumption that x_0 is the first variable that had a valid qualifier removed from its set of qualifiers.

Local Soundness of the transfer functions. There is a transfer function f_e for each expression e. Each transfer function f_e takes as input a set mapping S, and outputs an updated mapping S'. Let f_e be the transfer function that removes the invalid qualifiers from the set of qualifiers of each variable $x_i \in e$. After the application of f_e, for each variable $x_i \in e$, and each $<p_i, q_i> \in S'[x_i]$, there exists $<p_1, q_1> \in S'[x_1]$,..., $<p_{i-1}, q_{i-1}> \in S'[x_{i-1}]$, $<p_{i+1}, q_{i+1}> \in S'[x_{i+1}]$, ..., $<p_n, q_n> \in S'[x_n]$, such that $<p_1, q_1>$,...,$<p_n, q_n>$ type-check with the rule for e in Fig. 4.6. Making e type check requires that the typing rule for e holds.

$$
\begin{aligned}
&\forall x_i \in e, S'[x_i] = \{<p_i, q_i> | \\
&<p_i, q_i> \in S[x_i] \text{ and} \\
&\exists <p_1, q_1> \in S'[x_1], \ldots, <p_{i-1}, q_{i-1}> \in S'[x_{i-1}], \\
&<p_{i+1}, q_{i+1}> \in S'[x_{i+1}], \ldots, <p_n, q_n> \in S'[x_n] \text{ s.t.} \\
&<p_1, q_1>, \ldots, <p_n, q_n> \text{ type-check with the rule for } e\}
\end{aligned}
$$

Proposition 2. If. after n refinements, there is a valid typing T and the extracted object graph G based on T is sound.

Proof. (Sketch) For each variable x in the program, $T[x]$ maps to a qualifier $<p, q>$. For all variables x, the qualifier $<p, q> = T[x]$ type-checks the expression e where $x \in e$. Therefore, the typing T type-checks every expression in a program P, and based on the soundness of the object graph discussion in [3] the object graph is sound. Using abstract interpretation, the extraction analysis reads the qualifier $T[x]$ for each variable x and extracts a sound object graph. The soundness proof is left for future work.

4.8 Time Complexity

The state of the art of ownership inference, Huang et al. [21], claims a quadratic time complexity on the number of variables. At each iteration of the inference analysis, the set of qualifiers of at least of one of the variables of the program changes to a smaller set. Therefore, the upper bound time complexity of each iteration of the inference analysis is $\mathcal{O}(|Q_F|^2 * n)$, where n is the number of variables and Q_F is the full set of qualifiers of the type system. Another important point is that n counts the TAC variables, which are not actual variables in the program, but the set-based solution treats them the same as the variables declared in the program. Therefore, n is a multiple of the number of variables that are declared in the program. The size of Q_F for OT and UT are 3 and 6, respectively, which are small constants. The analysis may make as many iterations as there are variables (or finish sooner if it hits an empty set). So the overall time complexity of the set-based solution is $\mathcal{O}(|Q_F|^2 * n^2)$ and the upper bound time complexity is $\mathcal{O}(n^2)$.

Huang et al. instantiate their framework for OT and UT. Since the optimality property holds for UT, the time complexity for inferring UT qualifiers is quadratic. For OT, the time complexity is quadratic, multiplied by a constant C, which is the number of times developers run the analysis after adding qualifiers to resolve conflicts. Recall, in OT, developers make the sets for selected variables to be singletons by specifying the full pair $<p, q>$. The number of such annotations is around 2-10 per KLOC. If each refinement corresponds to changing only the owning domain of one qualifier, based on Huang et al. measures, 4-20 completed refinements may be needed per KLOC.

Upper bound time complexity for UT : $\mathcal{O}(n^2)$ *s.t.* n is number of variables

Upper bound time complexity for OT : $C \times \mathcal{O}(n^2)$ *s.t.* n is number of variables and C is the number of times that developers run the analysis before it can find a typing. C is roughly (2-10 * LOC(P)/1000) where LOC is the lines of code in program P.

In our approach and using SOD, the time complexity of running the inference analysis for each refinement is quadratic. The size of Q_F for SOD is at most 28, which is a relatively big constant and makes the running time of the set-based solution for SOD a factor of its running time for OT or UT. If the program contains n variables and the developers attempt m refinements, the time complexity to find a valid typing is $m \times \mathcal{O}(|Q_F|^2 * n^2)$. If we add the second ownership parameter, the upper bound for the size of Q_F would be 7^3 (three parameters including `owner` and two domain parameters). By adding the third ownership parameter, the size of Q_F would be 7^4 that significantly slows down each iteration of the analysis.

Upper bound time complexity for SOD, this work : $m \times \mathcal{O}(n^2)$ *s.t.* n is number of the variables m is the number of refinements

Compared to my Masters' thesis version of this work (Section 3.11), this is a big improvement. In the former system, the time complexity is exponential after each refinement, while running the transfer functions is also quadratic, the analysis has another phase where it tries to resolve all the remaining conflicts by enumerating all the possible combinations of the qualifiers for all the variables. If the program has n variables and each variable has l qualifiers in its set of qualifiers, then the time complexity for each refinement is $\mathcal{O}(l^n)$. So

for m attempted refinements, the overall time complexity is $m \times \mathcal{O}(l^n)$

Upper bound time complexity for SOD, MS work : $m \times \mathcal{O}(l^n)$ *s.t.* n is the number of variables and m is the number of refinements and l is the number of qualifiers for each variable

For the Auto and Assisted modes, the overall end-to-end time complexity is higher. For each automated refinement, the time complexity of running the adaptation and transfer functions is still $\mathcal{O}(n^2)$. If there are m variables as the target variables of the automated refinements, the worst case is to apply all the three kinds of refinements (MakeOnwedBy, MakePartOf and MakePeerWith) on all m variables. Therefore, the total time complexity is $\mathcal{O}(mn^2)$. The upper bound is $\mathcal{O}(n^3)$, since the refinements can be applied on all the variables in the program. For Assisted, the constant m would be lower, since some of the variables would be the target variables of the manual refinements, but the upper bound is the same.

4.9 Summary

In our approach, we go to great pains to never infer qualifiers that fail to type-check. To enforce these guarantees, we formalize the inference analysis using a core model of the language, and respect the type system typing rules. In fact, the transfer functions enforce the typing rules on the set mapping and for each set of qualifiers, with respect to the expression that the variable is used in. The typing rules type-check a typing that maps each variable to a single qualifier. Even though we model our approach for a subset of the language, we still formalize the adaptation for additional language features such as generic types that require the inner ownership parameter and inheritance. Supporting those features enables us to evaluate the analysis on real world subject systems. In the following chapter, we evaluate the approach using two subject systems.

CHAPTER 5 EVALUATION

In this chapter, we evaluate OOGRE using two subject systems. First, we explain the evaluation method, then we mention the pre-processing steps for each subject system. Next, we discuss the evaluation on the first subject system, JDepend. Then we discuss the evaluation on the second subject system, MiniDraw. We use "we", this dissertation's author, to refer to the developer conducting the evaluation.

5.1 Evaluation Method

Using the following steps, we evaluate each subject system. The steps are:

1. We pre-process the code of a subject system in preparation to run the inference and extraction analysis;
2. We follow the developer interaction steps explained in Section 3.7;
3. We apply refinements;
4. We measure how many refinements are attempted, done, skipped, and classify them by refinement type;
5. We measure the time of each done or skipped refinement;
6. We compute metrics on the inferred qualifiers of a typing;
7. We extract object graphs;
8. We visually compare hierarchical object graphs of a subject system with each other and with its flat object graph and highlight important differences;
9. We compute some predefined metrics on the object graphs and compare the results.

5.2 Code Pre-processing

We preprocess the code of a subject system as follows. Some steps are done using refactoring tool support in Eclipse, others are done manually.

1. Extracting local variables: the idea is to be able to add a qualifier at an object creation expression; our current implementation does not support specifying qualifiers directly

on object creation expressions;

2. Adding generic type arguments: the extraction analysis abstracts objects to pairs of type and domain; so it is useful to have precise declared types, including generic types;
3. Removing unnecessary casts after adding generic type arguments;
4. Replacing use of iterators with `for` loops: having only one ownership domain parameter, SOD is not able to express the Iterator design pattern;
5. Converting anonymous classes to top-level classes: the idea is to be able to attach declarations of `owner` and `p` domain parameters to a class declaration;
6. Adding missing default constructors: in the dataflow analysis framework, the transfer functions need the constructors in order to analyze declarations of fields of a non-primitive type;
7. Adding library stubs: the default qualifiers for library code lead to unsoundness (missing edges) in the object graph; we add library stubs for some of the classes in `java.lang`, `java.io` and `java.util` packages and declare virtual/ghost fields in the `java.util` collection classes;
8. Adding a synthetic `Main` class: the extraction analysis needs a single class that serves as the type of the root object, and declares the top-level domains;
9. The implementation produces one entry point for the application: the object graph extraction requires a single rooted object graph. For example, JDepend has 3 entry points: `swingui`, `xmlui` and `textui`; we pick `swingui`;

5.3 JDepend Evaluation

The first subject system we evaluate is called JDepend, and is 4766 LOC. We do not know much about this system, so we first use the Auto mode to extract a first object graph. By studying the object graph, we gain an understanding of the design intent of the system and the interactions between the important objects. Since the Auto mode does not quite express our design intent, we start over using the Manual mode to express our design intent. In this section, we list the summary of the refinements from which Auto and Manual modes

infer a typing. We also measure the qualifiers inferred by Auto and Manual. Separately, we reproduce the Huang et al. experiment using OT for the same system. We compute metrics on the object graphs of the three experiments and compare the results. Section 5.3.4 shows the details of each manual refinement and the object graphs.

5.3.1 System Overview

JDepend traverses packages of a Java project to generate metrics, such as number of classes and interfaces, number of packages that depend upon the classes of a package (Afferent Coupling), and number of packages that classes of a package depend on (Efferent Coupling). So the architecturally relevant classes are `JavaClass`, `JavaPackage`, and their dependent classes such as `PackageComparator`, `JavaClassBuilder` and `ClassFileParser`. The classes `AfferentNode` and `EfferentNode` express the coupling relationships.

5.3.2 Auto Mode

We follow the developer interaction in Section 3.7 for the Auto mode, and OOGRE finds a typing in 55 minutes (First two rows of Table 5.1). The object graph helps us to understand the system. One drawback of the Auto mode is some objects may not be placed in a domain that we want. For example, the object of type `Analyzer` is strictly encapsulated in the `JDepend` object, so it is not visible at the top level. However, the object of type `Analyzer` is the key logic of the application, and we want to see that object in a top-level domain.

5.3.3 Summary of Attempted Refinements

Using the Manual mode, we attempt 28 refinements in total to express our design intent and resolve conflicts, and out of those, 24 are done. At first, we apply 16 refinements to express our design intent, and out of those 14 are done. However, those refinements are not enough for OOGRE to find a typing, so we apply 14 more refinements to resolve the remaining conflicts, out of which 12 are done. We summarize the attempted refinements in the last two rows of Table 5.1.

Table 5.1: Attempted and done refinements using Auto and Manual modes. A refinement can be MakeOwnedBy (MOB), MakePartOf (MPO), MakePeerWith (MPW) or MakeShared (MSH). For the Manual mode, a (D) indicates that the refinement expresses design intent and a (C) indicates that the refinement resolves a conflict. #Mins is the average time in minutes for each refinement.

OOGRE mode	**Total #refinements**	**#Each type of refinement**				**#Mins**
		MOB	MPO	MPW	MSH	
Auto-Attempted	372	183	109	80	0	0.14
Auto-Done	169	74	29	66	0	0.32
Manual-Attempted	16 (D) + 12(C)	12	4	11	1	1.36
Manual-Done	14 (D) + 10(C)	9	3	11	1	1.57

5.3.4 Details of Manual Refinement Using the Manual Mode

We list here the 28 refinements that we do to express the design intent and to resolve conflicts using the Manual mode. A "-TLD" suffix after a refinement means its destination domain is a top-level domain:

1. **MakePeerWith**(`Analyzer`, `JDepend`): By looking at the object graph resulting from the Auto experiment, we understand that the object `analyzer` of type `Analyzer` is an architecturally-relevant object that has to be in a top-level domain. Therefore, we make it peer with the object of type `JDepend`; as the first refinement, there are many infeasible qualifiers that the analysis removes from the sets of qualifiers of the variables, so this refinement takes longer than the other refinements;
2. **MakePeerWith**(`AfferentNode`, `JDepend`): Another important object in JDepend is the object of type `AfferentNode` that we want also in a top-level domain; so we make the object of type `AfferentNode` peer with the object of type `JDepend`;
3. **MakeOwnedBy**(`JPanel`, `JDepend`): Our design intent is that the objects related to the Swing user interface should not be placed at the top level of the object graph, so we make this UI object of type `JPanel` strictly encapsulated;
4. **MakePeerWith**(`JavaPackage`, `JDepend`): Another architecturally-relevant object of the application is of type `JavaPackage` and we want to see it at the top level of the object graph, i.e., peers with other architecturally-relevant objects; therefore, we make the object of type `JavaPackage` peer with the object of type `JDepend`;
5. **MakePeerWith**(`DependTree`, `JDepend`): We apply this refinement to express the

design intent that the object of type `DependTree` must go hand in hand with the other objects, such as the objects of type `JDepend`, `AfferentNode` and `JavaPackage` at top level of the object graph;

6. **MakePeerWith**(`JavaClassBuilder`, `Analyzer`): Another important object is the object of type `JavaClassBuilder` and we want to see it at the top level of the object graph, i.e., peers with other important objects; so, we make the object of type `JavaClassBuilder` peers with the other important objects;
7. **MakeOwnedBy**(`Constant`, `ClassFileParser`): We do not want to see less architecturally-relevant objects at the higher levels of the object graph, so we make the object of type `Constant` strictly encapsulated in the object that creates it;
8. **MakeOwnedBy**(`PropertyConfigurator`, `Analyzer`): By looking at the object graph from Auto, we see that one of the less important objects of type `PropertyConfigurator` is at the same level as the important object of type `Analyzer`; so to unclutter the top level of the object graph, we apply this refinement, but it is skipped since an alias to the object of type `PropertyConfigurator` is passed to another object via a public method;
9. **MakePeerWith**(`PropertyConfigurator`, `Analyzer`): We apply this refinement to make the object of type `PropertyConfigurator` peers with the object of type `Analyzer`;
10. **MakeOwnedBy**(`ArrayList<String>`, `Analyzer`): We express the design intent that the data structure objects should be in lower levels of the object graph; so we make the object of type `ArrayList<String>` encapsulated in the object of type `Analyzer`;
11. **MakePartOf**(`ArrayList<JavaClass>`, `JavaClassBuilder`): To move objects of a data structure type in the lower levels of the object graph, we make the object of type `ArrayList<JavaClass>` strictly encapsulated, but we know that there is a pubic method in the code that returns an alias to that object, so it cannot be strictly encapsulated; therefore, we apply this refinement, but it is also skipped due to a missing

`final` field;

12. **MakeOwnedBy**(`GridLayout`, `JDepend`): By looking at the resulting object graph of Auto and to unclutter its top-level, we make the UI object of type `GridLayout` strictly encapsulated;
13. **MakePartOf**(`JarFile`, `JavaClassBuilder`): The next refinement is to express the design intent that one object is part of another one; we express that the object of type `JarFile` is logically contained in the object of type `JavaClassBuilder`;
14. **MakeOwnedBy**(`PackageComparator`, `JDepend`): By looking at the resulting object graph of Auto, we make the object of type `PackageComparator`, which is not architecturally relevant, strictly encapsulated in the object of type `JDepend`;
15. **MakePeerWith**(`AfferentNode`, `AfferentNode`): We want to see the object of type `AfferentNode` at the top level of the object graph; that object creates another object of the same type and to make them peers at the top level, we apply this refinement;
16. **MakePartOf**(`PackageComparator`, `PackageComparator`): To express the design intent of the class `PackageComparator`, which implements the Composite design pattern, we make the object of type `PackageComparator` part of itself by applying this refinement; This is the classic example of the Composite design pattern in which an object is part of an object of the same type;
17. **MakeOwnedBy**(`PropertyConfigurator`, `PackageFilter`): At this point we are done with the refinements that express our design intent, but OOGRE still cannot find a typing; so we open the MoreInfoNeeded window to investigate the conflicts and do refinements to resolve them. The conflict is the expression `config.getFilteredPackages();` in the class `PackageFilter`. We apply this refinement by looking at the set of qualifiers of the variables of the conflict expression. OOGRE is still unable to find a typing and there are more conflicts to resolve;
18. **MakeOwnedBy-TLD**(`JDepend`, `Main`): The next conflict is the method invocation expression `jDepend.instanceMain(args);` in the `Main` class; this is a conflict at the

top-level domain and we apply this refinement to resolve the conflict; we know that the `owned` domain of the `Main` class does not have the properties of a private domain, but we do this refinement to resolve the conflict;

19. **MakeOwnedBy**(`DataInputStream`, `ClassFileParser`): The next conflict is the expression `new DataInputStream(is);` in the class `ClassFileParser`; by looking at the set of the qualifiers of the variables of the conflict, we apply this refinement to resolve the conflict;
20. **MakeShared**(`fileInputStream`): The next conflict is `parser.parse(fis);` in the class `JavaClassBuilder`; We know that the object of type `FileInputStream` is from the library and it is intended to be shared between different objects; therefore, we apply this refinement and it resolves the conflict;
21. **MakePeerWith**(`FileManager`, `JavaClassBuilder`): Another conflict to resolve is `fileManager.acceptJarFile(file12)`; we make the `FileManager` object peers with the `JavaClassBuilder` object;
22. **MakePartOf**(`FileInputStream`, `PropertyConfigurator`): The next conflict to resolve is `new FileInputStream(file);` in the class `PropertyConfigurator`; the left-hand side of the object creation expression contains qualifiers with `PD` as their owning domain in its set of qualifiers, so we make the `FileInputStream` part of the object of type `PropertyConfigurator`;
23. **MakeOwnedBy**(`AttributeInfo`, `ClassFileParser`): The next conflict to resolve is the method invocation `result.setValue(value);` in the class `ClassFileParser`; the receiver, `result` is of type `AttributeInfo`, so we apply this refinement to resolve the conflict;
24. **MakeOwnedBy**(`ArrayList<ParserListener>`, `ClassFileParser`): The next conflict to resolve is the method invocation `parseListeners.add(listener);`; the receiver ,`parseListeners` is of type `ArrayList<ParserListener>`; to resolve this conflict we should apply a refinement with the destination object of type `AbstractParser`,

but `AbstractParser` is an abstract class and does not have a representative node in the object graph; so we apply this refinement, since `ClassFileParser` is a subclass of `AbstractParser`; however, the refinement is skipped;

25. **MakeOwnedBy**(`ArrayList<ParserListener>`, `Parser`): To resolve the same conflict, we change the destination object of the previous refinement to be `Parser` as the other subclass of `AbstractParser`, but this refinement is also skipped;
26. **MakePeerWith**(`JDependParserListener`, `JDepend`): The two previous refinements to resolve the conflict are skipped, so we do a refinement on the elements of the list of type `ParserListener`to resolve the conflict; `ParserListener` is an Interface type and we do the refinement on one of the classes that implements it, `JDependParserListener`; we apply this refinement and it resolves the conflict, but there is another conflict to resolve;
27. **MakePeerWith**(`AfferentNode`, `DependTree`): Another conflict to resolve is `new AfferentNode(null, javaPackage);` in the class `DependTree`; our design intent is that the objects of type `AfferentNode` and `DependTree` should be peers at the top level of the object graph, so we make those objects peers, but OOGRE reports the same conflict expression;
28. **MakePeerWith**(`JavaPackage`, `DependTree`): OOGRE reports the same conflict, but now the problem is the set of qualifiers of the argument of the constructor, `javaPackage` of type `JavaPackage`; our design intent expresses that the object of type `JavaPackage` should also be peer with the object of type `DependTree`; so we apply this refinement and it resolves the conflict; after this refinement, the analysis finds the typing that expresses our the design intent and OOGRE extracts the object graph.

The summary of the refinements is shown in Table 5.2. For each of the 28 refinements, we show the type of the refinement with a (D) if the refinement expresses design intent or a (C) if the refinement resolves a conflict. For example, refinement 1 is a MPW(D) i.e., we use MakePeerWith to express the object of type `Analyzer` be peer with the ob-

ject of type `JDepend`. The remaining refinements express code idioms and design patterns e.g., refinement 16, MPO(D) expresses the Composite design pattern for the object of type `PackageComparator` by making it part of itself. Next, we apply refinements to resolve conflicts, for example, using the refinement 20, which is a MSH(C), we resolve a conflict by placing the object of type `FileInputStream` in the domain `shared`.

Table 5.2: The refinements on JDepend to find a typing. The first column is the type of a refinement (MakeOwnedBy (MOB), MakePartOf (MPO), MakePeerWith (MPW) and MakeShared (MSH), prefix "S-" means the refinement is a SplitUp, suffix "-TLD" means the destination domain of the refinement is a top-level domain) (D) indicates that the refinement expresses design intent, (C) indicates that the refinement resolves a conflict. The rest of the columns are: type of the source and the destination objects of a refinement, the analysis running time to apply each refinement in minutes and if the refinement succeeded or not.

No.	Ref. (D/C)	Source Type	Destination Type	#Mins	Status
1	MPW(D)	`Analyzer`	`JDepend`	7.56	Done
2	S-MPW(D)	`AfferentNode`	`JDepend`	2.83	Done
3	MOB(D)	`JPanel`	`JDepend`	2.78	Done
4	S-MPW(D)	`JavaPackage`	`JDepend`	3.32	Done
5	MPW(D)	`DependTree`	`JDepend`	0.53	Done
6	S-MPW(D)	`JavaClassBuilder`	`Analyzer`	0.48	Done
7	MOB(D)	`Constant`	`ClassFileParser`	1.40	Done
8	S-MOB(D)	`PackageConfigurator`	`Analyzer`	0.15	Skipped
9	S-MPW(D)	`PackageConfigurator`	`Analyzer`	0.50	Done
10	S-MOB(D)	`ArrayList<String>`	`Analyzer`	1.07	Done
11	MPO(D)	`ArrayList<JavaClass>`	`JavaClassBuilder`	0.15	Skipped
12	MOB(D)	`GridLayout`	`JDepend`	0.48	Done
13	MPO(D)	`JarFile`	`JavaClassBuilder`	0.93	Done
14	MOB(D)	`PackageComparator`	`JDepend`	1.09	Done
15	S-MPW(D)	`AfferentNode`	`AfferentNode`	2.06	Done
16	S-MPO(D)	`PackageComparator`	`PackageComparator`	0.87	Done
17	S-MOB(C)	`PropertyConfigurator`	`PackageFilter`	1.35	Done
18	MOB-TLD(C)	`JDepend`	`Main`	0.70	Done
19	MOB(C)	`DataInputStream`	`ClassFileParser`	2.86	Done
20	S-MSH(C)	`FileInputStream`	`shared`	0.52	Done
21	S-MPW(C)	`FileManager`	`JavaClassBuilder`	1.45	Done
22	S-MPO(C)	`FileInputStream`	`PropertyConfigurator`	0.74	Done
23	MOB(C)	`AttributeInfo`	`ClassFileParser`	1.99	Done
24	MOB(C)	`ArrayList<ParserListener>`	`ClassFileParser`	0.10	Skipped
25	MOB(C)	`ArrayList<ParserListener>`	`Parser`	0.10	Skipped
26	MPW(C)	`JDependParserListener`	`JDepend`	0.94	Done
27	S-MPW(C)	`AfferentNode`	`DependTree`	0.60	Done
28	S-MPW(C)	`JavaPackage`	`DependTree`	0.61	Done

5.3.5 Metrics on the Inferred Qualifiers

Table 5.3 measures the qualifiers of the inferred typings by OOGRE in the Auto and Manual modes. For the reproduced experiment, we show its numbers (OT-repro), as well

as the original numbers (OT-orig). We reproduce the OT experiment by translating OT qualifiers into the SOD subset (i.e., not using `PD`, `lent` or `unique`) to be able to extract the object graph and compute metrics on it. In reproducing the experiment, we keep the explicit qualifiers they provide[1] and supply remaining ones that respect the OT ranking. The OT-repro numbers are slightly different from the OT-orig ones because of the pre-processing steps (Section 5.2, e.g., isolating and analyzing only the `swingui` entry point) and the library stubs that we add. So the number of variables is the same for Auto, Manual and OT-repro and different from Huang et al. for OT-orig and UT.

When discussing Table 5.3 below, we compare the owning domain of the inferred qualifiers. We refer to them simply as modifiers. By comparing the modifiers of Auto and Manual, Auto tends to respect the ranking more. So the number of the local domain modifiers (`owned` or `PD`) is higher in the typing that Auto infers. This is unsurprising since Auto attempts MakeOwnedBy first to find all the strictly encapsulated objects. For the objects that are not encapsulated, Auto attempts MakePartOf on their corresponding variables. Finally, if an object is neither strictly encapsulated nor logically contained, Auto attempts MakePeerWith. Therefore, more objects are placed in `owned`, `PD`, or `owner` compared to Manual. Using the Manual mode, we express our design intent, which can be different from the results of Auto. If we want to see an object at top level of the object graph, then we do not attempt MakeOwnedBy on it, instead we use a MakePeerWith. For this system, we did not think carefully about the top-level domains and how to split objects in them. So we put all the objects in one top-level domain (`owned`), which leads to no `p` modifiers. The analysis infers `owner` and does not use `p`, if we place objects in the same top-level domain. If we split objects across two top-level domains, the analysis infers `p`, like in the typing inferred by Auto that contains 32 `p` modifiers. If we were to redo the evaluation, we would split the objects in two top-level domains.

Comparing Auto and OT-repro, Auto infers 19 more `owned` modifiers, because it exhaus-

[1] http://www.cs.rpi.edu/ huangw5/cf-inference/

tively attempts to make every variable `owned`. Also Auto infers 59 `PD` modifiers that OT does not support. Those modifiers are mostly replaced with `owner` and `p` in OT-repro. As a result, the object graph of Auto is more hierarchical than the one from OT-repro (Look at Table 5.5 for metiers on the object graphs).

Manual infers fewer `owned` modifiers compared to OT-repro. Our design intent is to place the architecturally relevant objects at the higher levels of the hierarchy, not make as many objects as possible to be encapsulated. Also OT forces objects to be either owned or peers, but with SOD objects can be in a public domain as shown by the 33 `PD` modifiers.

For Huang et al.'s UT experiment, the optimality property holds and there is no need to resolve conflicts. However, there are only 14 `owned` modifiers out of 542. Instead, there are 433 `owner` modifiers (really `peer` in UT) that make many objects peers, so the object graph does not have much hierarchy. Moreover, there are 95 `any` modifiers that provide no ownership information and 102 purity modifiers that must be separately inferred.

We compare the number of operations such as automated refinements, manual refinements, or manually added qualifiers across the experiments (Table 5.4). Auto attempts 319 automated refinements. Using the Manual mode, we apply 28 manual refinements. In OT-repro, we add 21 qualifiers manually, compared to Huang et al. who add 26 qualifiers manually.

The qualifiers do not directly describe the structure of the object graph. For example, it is hard to estimate the depth of architecturally-relevant objects or low-level objects in the object graph from the qualifier of their corresponding variables. So we extract object graphs and compute metrics on them for the Auto, Manual and OT-repro experiments.

5.3.6 Object Graphs

Fig. 5.1 shows the object graph for Auto, Fig. 5.2 illustrates the object graph for Manual and the object graph for OT-repro is Fig. 5.4. We show only the top-level objects expanded. We expect to see more architecturally-relevant objects from the application at the top level of each graph rather than low-level objects that are data structures such as `ArrayList` and

Table 5.3: Metrics on the qualifiers for the different experiments. **vars** is the number of variables. Each column named with a modifier indicates the number of qualifiers with that modifier as the owning domain. For example, column `owned` indicates the number of <`owned`,_> qualifiers. The column `<p>` is the number of qualifiers in that form, for variables that are type parameters. The columns `any` and `pure` are only applicable to UT and indicate the number of `any` and the number of `pure` qualifiers that are added to the code, respectively.

Experiment	**vars**	`owned`	`PD`	`owner`	`p`	`shared`	`<p>`	`lent`	`unique`	`any`	`pure`
Auto	562	122	59	110	32	131	46	34	28	n/a	n/a
Manual	562	73	33	110	0	165	46	61	74	n/a	n/a
OT-repro	562	103	n/a	168	119	123	46	3	n/a	n/a	n/a
OT-orig [21, Fig. 10]	542	130	n/a	156	128	128	n/a	n/a	n/a	n/a	n/a
UT [21, Fig. 9]	542	14	n/a	433	n/a	n/a	n/a	n/a	n/a	95	102

Table 5.4: Total number of operations to find a typing for each experiment.

Experiment	**Operations**	**Notes**
Auto	319	automated refinements; no input from developers
Manual	28	manual refinements; developers indicate p in a qualifier
OT-repro	21	manual qualifiers; developers indicate both p and q in a qualifier
OT-orig [21, Fig. 10]	26	manual qualifiers; developers indicate both p and q in a qualifier
UT [21, Fig. 9]	0	no input from developers since the optimality property holds

`Hashtable` for most applications.

Thanks to different types of manual refinements, Manual enables us to express our design intent by showing all the important objects at the top level. We see objects of type `JDepend`, `Analyzer`, `JavaPackage`, `EfferentNode`, `AfferentNode` and their relations at the top level. Also we do not see `swing` UI objects or data structures at the top level. Using Manual, we manage to move most of those less important objects to the lower levels of the graph and the remaining ones cannot be moved further down, because the code does not support the refinements. On the other hand, owing to the ranking and the ability to express logical containment (which is not present in OT), Auto manages to find all the strictly encapsulated objects and all the object that can be logically contained in the object graph, so it splits objects between the two top-level domains. Therefore, the resulting object graph is very hierarchical and its maximum depth is 36 (Table 5.5). Even though there are objects that we want to see at the top level such the object of type `Analyzer`, Auto, however, moves them down to lower levels of the object graph.

5.3.7 Analysis of the Object Graphs

We show the flat object graph of JDepend in Fig. 5.3. By comparing the flat object graph with the hierarchical object graphs from each experiment, we notice that using OOGRE, the hierarchical object graphs are more readable and useable. In addition to comparing hierarchical object graphs with the flat one, we analyze the hierarchical object graphs to understand what we can learn from each one.

By looking at the object graph of the Auto experiment (Fig. 5.1), we notice that this object graph has five objects in the top-level domains. Although it is a huge improvement over the flat object graph, this object graph does a poor job of showing important objects of the application and their communication. To use this object graph, developers have to drill down to find the important objects and to understand how they interact with the other objects. Although the object graph is hierarchical, it is still big and finding a specific object can be hard. We address this issue using the Manual experiment, by moving the architecturally relevant objects to the top-level domains.

Using the Manual experiment, we have the flexibility to move some of the important objects to top-level domains of the object graph to highlight their communication with other objects. We do refinements on the objects of type `Analyzer`, `EfferentNode`, `AfferentNode` and `DependTree` to move them to the top level (highlighted in the object graph, Fig. 5.2). By looking at the top level of the object graph, we understand that the object of type `JDepend` points-to an object of type `DependTree`, which has a `model` of type `DependTreeModel`. The `model` object has `EfferentNode` and `AfferentNode` objects that share an object of type `JavaPackage` (highlighted path/thick edges in the object graph). The object graph from the Auto experiment looks less busy and more concise, but by showing important objects and their relations, the object graph from the Manual experiment is more useful for understanding the system.

Having the hierarchical object graphs from each experiment, the question is: can we make the object graph from the Manual experiment more hierarchical? There are some objects

in the `shared` domain, and there are some object without any edges, let us see if we could move those objects from the `shared` domain to some other domains. For the objects of type `String`, `Integer`, `Double`, `Float` and `Long`, we could use the notion of manifest ownership and move them to the `shared` domain. We could hide the `shared` domain and all of its objects, since there is not much reasoning about objects in `shared`. However, we do not want to make the object graph unsound, so we show the `shared` domain. The object graphs currently show only points-to edges. If we were to show some other kinds of edges like data flow edges, we would likely see edges for the objects with no points-to edges. Another object that we see in the `shared` domain is the object of type `FileInputStream`, for which we do a MakeShared refinement (refinement number 20 in Table 5.2).

The approach may require additional kinds of refinements to make the final graphs even more abstract. Abstraction by type is one kind of refinement that merges two or more objects of some distinct subclasses of a super class into an object of the type of the super class [3]. Abstraction by type can be readily added to OOGRE since it involves changing how the graph is displayed and requires no changes to the inference analysis. However, for JDepend, there are not many inheritance hierarchies and subclasses, so it is not likely to help here. Another kind of refinement that could make the final object graphs more abstract is abstraction by name [35].

In the object graph of the OT-repro experiment(Fig. 5.4), we highlight the objects that are placed in the top-level domain `owned` of the object, but they are not in the top level of the object graph of the Manual experiment. There are 4 data structure objects and one low-level object of type `File` (highlighted with a solid filled arrow points to each one) that are placed in the top-level `owned`, but we mange to move them in lower levels using Manual. We are able to move those objects that are not strictly encapsulated in `PD`, but using OT-repro, they have to stay peers with the objects that create them. Moreover, there are 2 `swing` UI objects (highlighted with an empty-fill arrow pointing to each one), which are placed in the top-level `owned` in the object graph of OT-repro, but they are not at the top level of the object graph

of the Manual experiment. There is one data structure object of type `ArrayList<Byte>` (highlighted with an empty-fill dashed border arrow pointing to it), which is in the top-level `owned` for OT-repro, but for Manual, that object is in `shared`, which is caused by the MakeShared refinement that we do on the object of type `FileInputStream`.

5.3.8 Metrics on the Object Graphs

We compute some previously defined metrics [35] on the object graphs of the three experiments and the results are in Table 5.5. The metrics are as follows:

1. **Number of Objects (#O):** the number of all the objects in the object graph;
2. **Top-Level Objects (#TLO):** the number of objects at the top-level domains;
3. **Number of Low-Level Objects in Top-Level Domains (#LLO):** a low-level object is an object that the developer does not want to see it at top level of the object graph, e.g., data structure objects;
4. **Objects in PD (#OPD):** the number of objects in the public domain (`PD`);
5. **Objects in PrD (#OPrD):** the number of objects in the private domain (`owned`);
6. **Object Depth (OD):** the object depth, including the Average (Avg OD), Minimum (Min OD) and Maximum (Max OD);
7. **Maximum Depth of Ownership Hierarchy (MXD):** the longest path length from the root object in the object graph.

Using Auto, since the extraction analysis keeps objects that are in different domains as distinct, #O is higher than the other experiments, due to many refinements that Auto automatically applies. Moreover, Auto has lower #TLO because it aggressively pushes objects down to get more hierarchy. If an object cannot be in `owned`, it can be in `PD`. OT-repro has the highest #TLO since it does not support public domains and has to make objects peers. In Manual, we express our design intent to see important objects at the top level, so its #TLO is higher compared to Auto. Unlike OT which has only owned or peer, with SOD, we can still push objects down by placing them in PD using Manual, so its #TLO is lower than OT-repro. The #OPD is zero for OT-repro, so OT cannot group objects of

the same hierarchy level in different domains and they have to be peers. The #OPrD is higher for OT-repro, because private domains (`owned`) are the only type of hierarchy that OT supports, but for SOD, these objects will be scattered across `owned` or `PD`. Also #OPrD is higher for Auto compared to Manual, because of MakeOwnedBy refinements that Auto applies exhaustively before any other type of refinement. The metrics related to depth of hierarchy (MXD and AOD) are relatively the same, but the MAX OD for Auto is higher than the others, since it exhaustively attempts all possible refinements and respects the ranking. The Max OD is higher for OT-repro compared to Manual, since our goal when using Manual is not to maximize hierarchy but to simply express our design intent.

Table 5.5: Object graph metrics extracted from the inferred typings across the three experiments.

Experiment	**#O**	**#TLO**	**#LLO**	**#OPD**	**#OPrD**	**MXD**	**AOD**	**min OD**	**max OD**
Auto	124	5	0	41	82	5	0.54	0	36
Manual	89	18	0	40	48	4	0.52	0	18
OT-repro	107	26	0	0	106	5	0.65	0	26

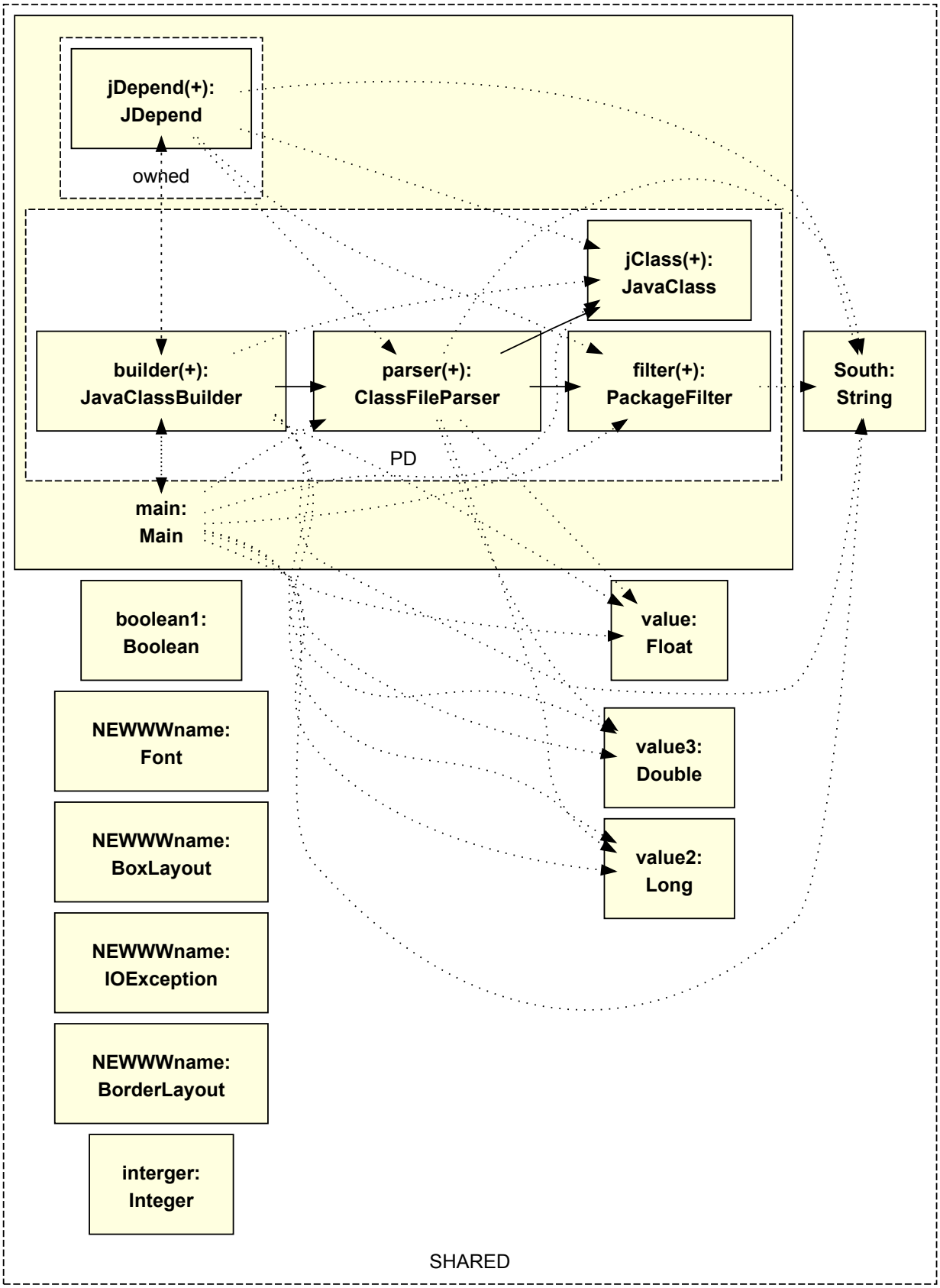


Figure 5.1: The object graph extracted from the typing for the Auto experiment.

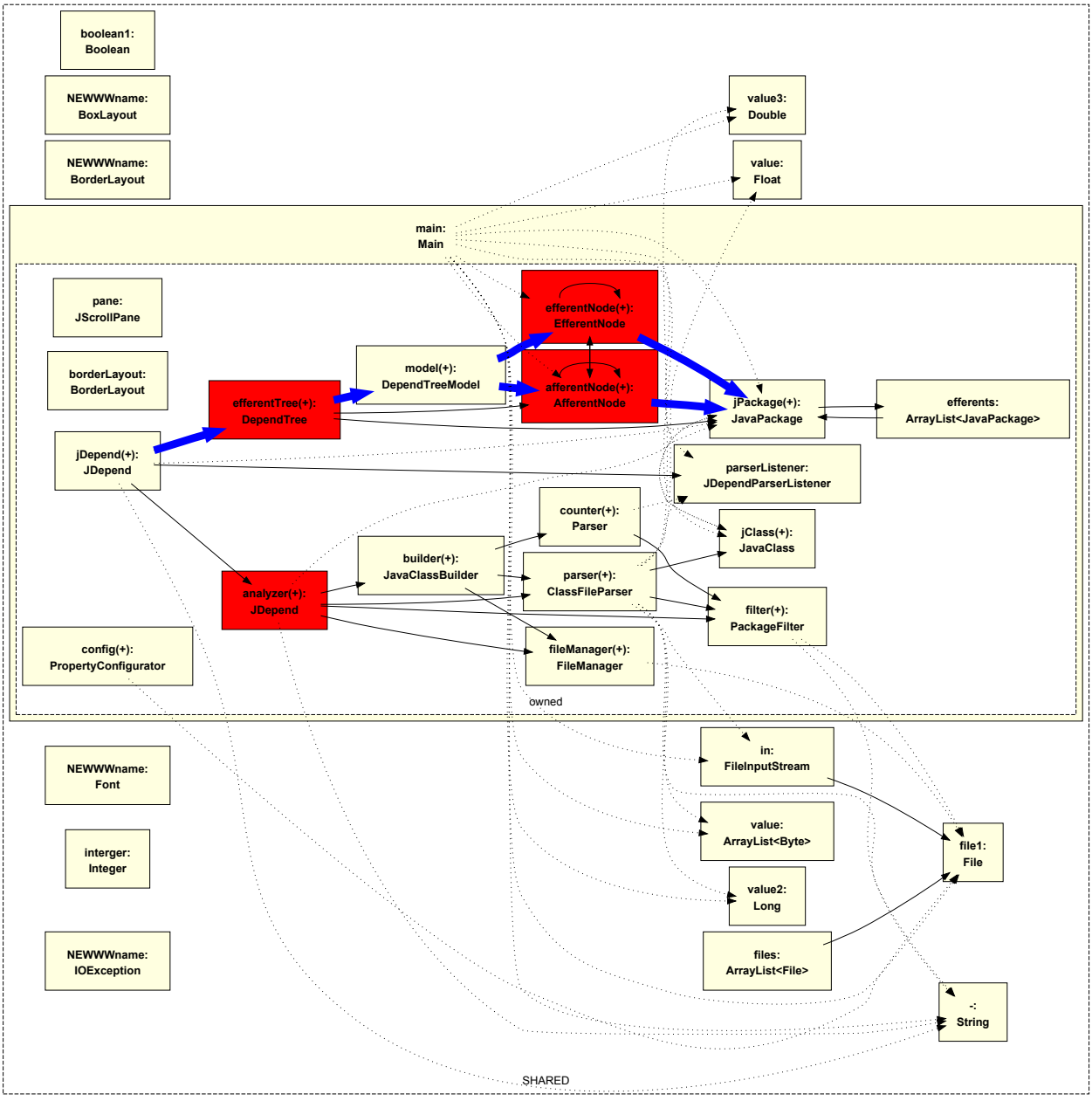


Figure 5.2: The object graph extracted from the typing for the Manual experiment.

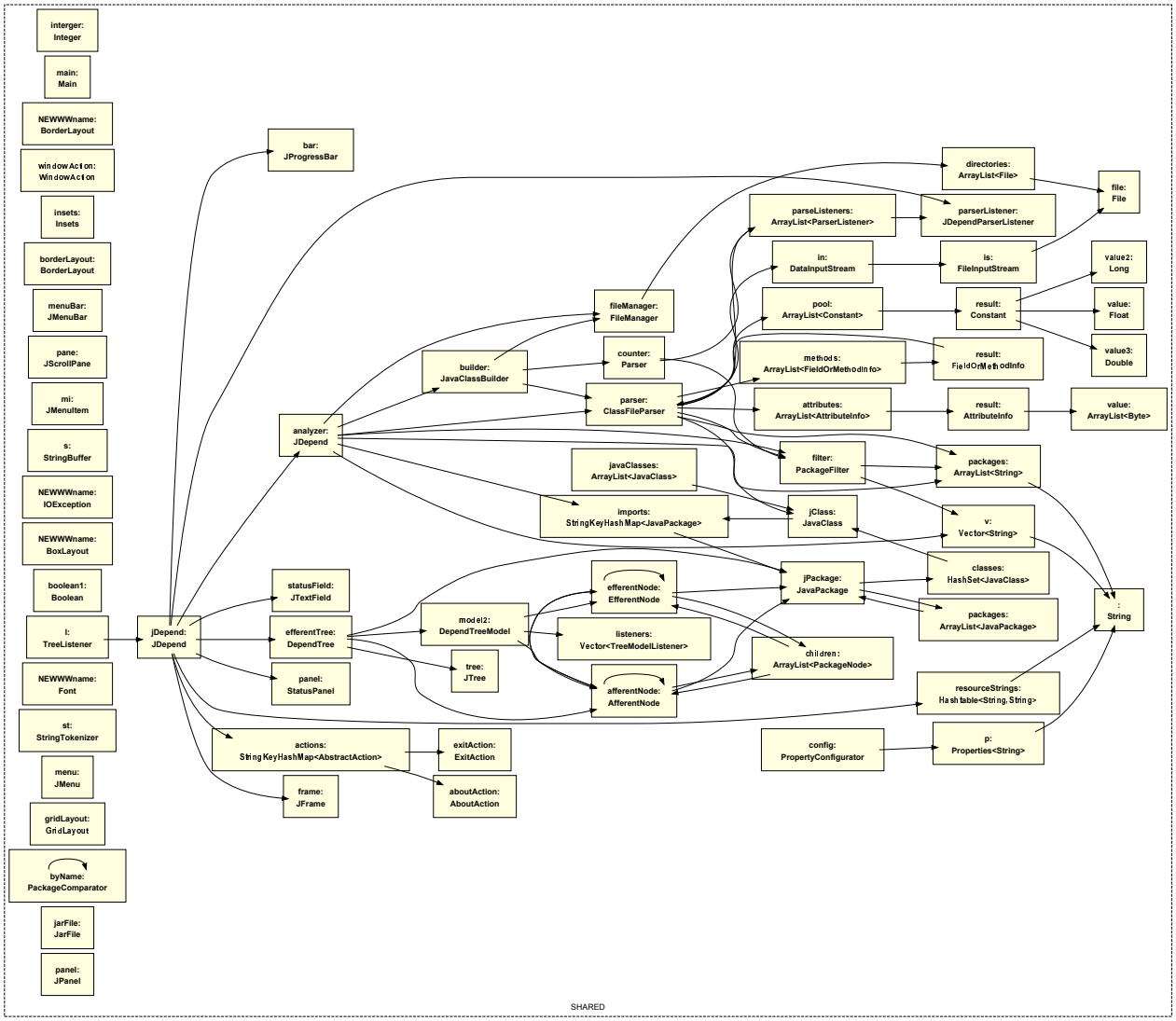


Figure 5.3: The flat object graph extracted from the default qualifiers for JDepend.

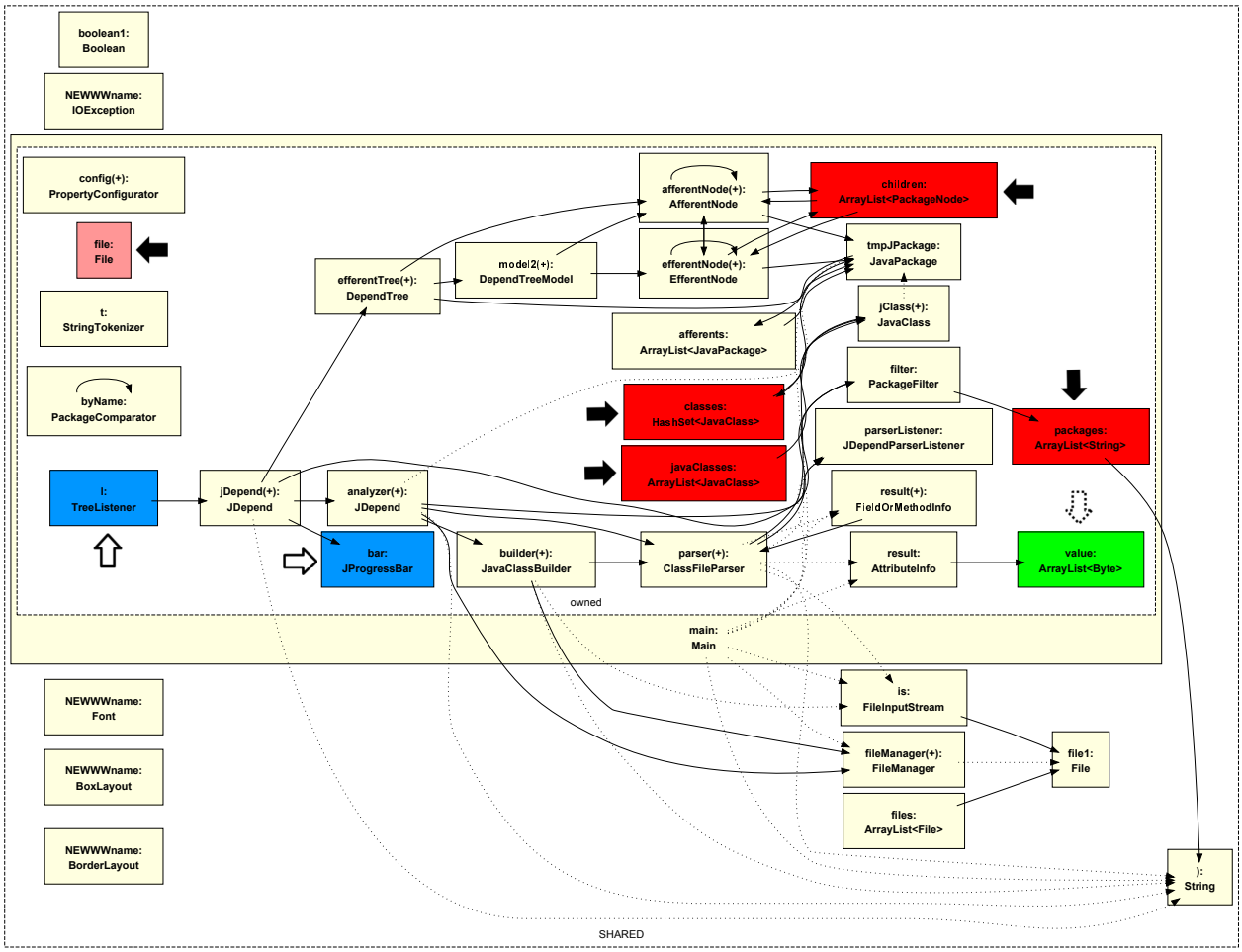


Figure 5.4: The object graph extracted from the typing for the reproduced OT experiment.

5.4 MiniDraw Evaluation

For our next system, MiniDraw (MD) (1.4 KLOC), we study its documented class diagrams, propose some refinements that express our design intent, then attempt them using OOGRE. We then evaluate if the extracted object graphs express our design intent.

5.4.1 System Overview

MD is an object-oriented framework for building graphical board game applications. We analyze one board game application, BreakThrough, built using MD. MD follows good object-oriented practice such as programming to an interface (has 17 interfaces), and polymorphism. MD also implements several design patterns such as Composite and Command.

MD follows the standard framework layering: core interfaces, default implementation classes of those interfaces, and utility classes. The `framework` package is a framework for graphical applications. The `boardgame` package is a framework to build a board game. MD uses the `minidraw` framework to build the BreakThrough board game and the `breakthrough` package contains the logic of the BreakThrough board game.

5.4.2 Summary of Refinements of Auto and Assisted Modes

Auto finds a typing for MD after 21 minutes. Although we set out to express the design intent from the available documentation, we still run Auto to recognize in the abstractions extracted from the code the architecturally relevant objects highlighted in the documentation. OOGRE finds 166 fields and local variables in the code and applies on them MakeOwnedBy, MakePartOf and MakePeerWith automated refinements. Table 5.6 shows the number of attempted and done refinements.

Using the Assisted mode, OOGRE finds 162 fields and local variables to attempt refinements on. After 19 minutes, when Assisted attempts 331 refinements out of which 104 are done, OOGRE finds a typing. Table 5.6 shows the done and attempted refinements using the Assisted mode.

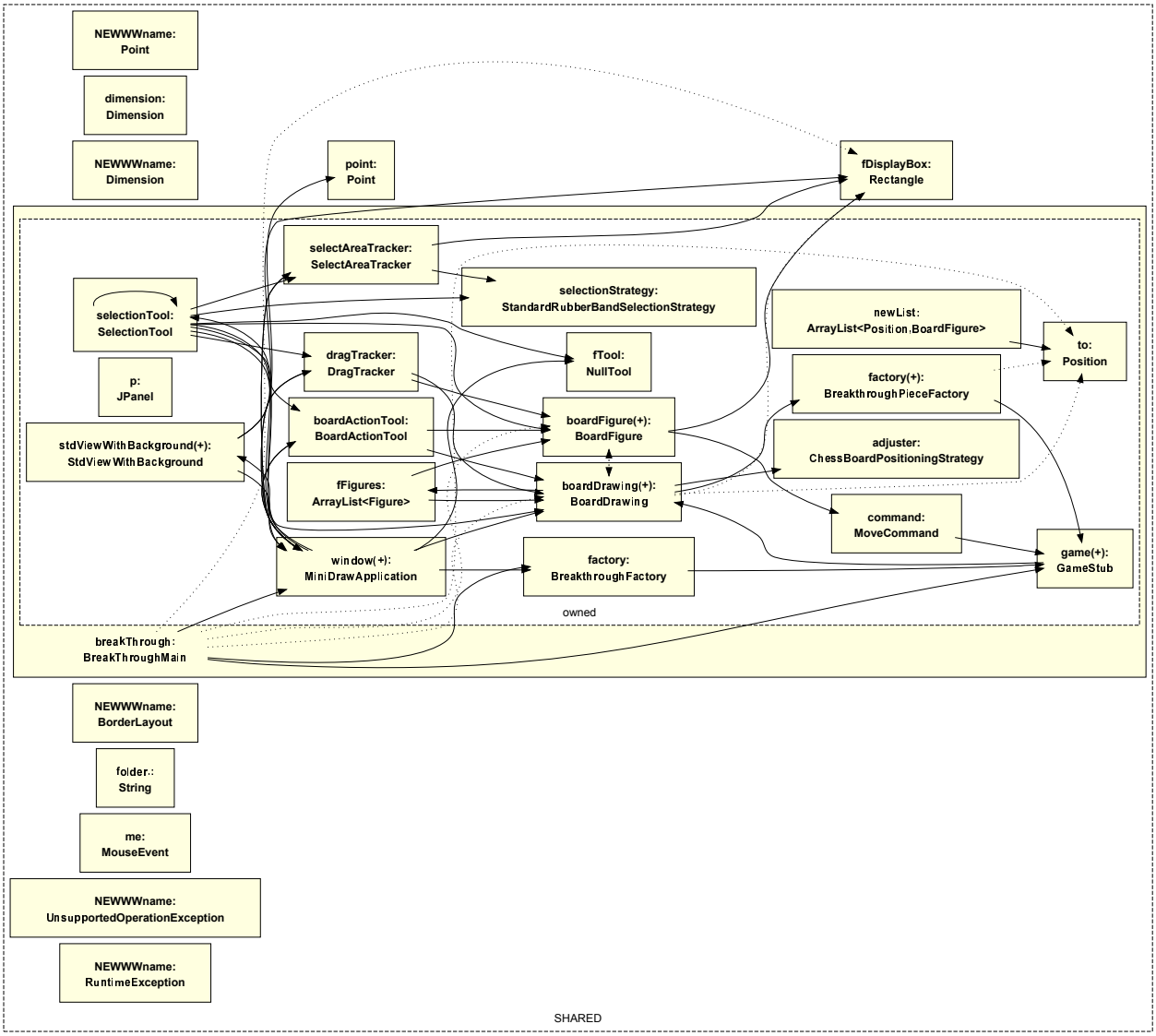


Figure 5.5: The top-level domains of the object graph for MD using the Auto mode. The root object is expanded. The (+) on an object label indicates a collapsed object sub-structure.

Table 5.6: Attempted and done automated refinements by Auto and Assisted modes on MD.

	Auto				**Assisted**			
	MOB	MPO	MPW	Total	MOB	MPO	MPW	Total
Done	51	6	75	132	46	5	53	104
Attempted	115	109	34	258	162	111	58	331

5.4.3 Procedure To Identify Refinements

We study the documentation and identify refinements as follows:

1. We look for different types of relations (association and aggregation) between classes in the class diagrams;

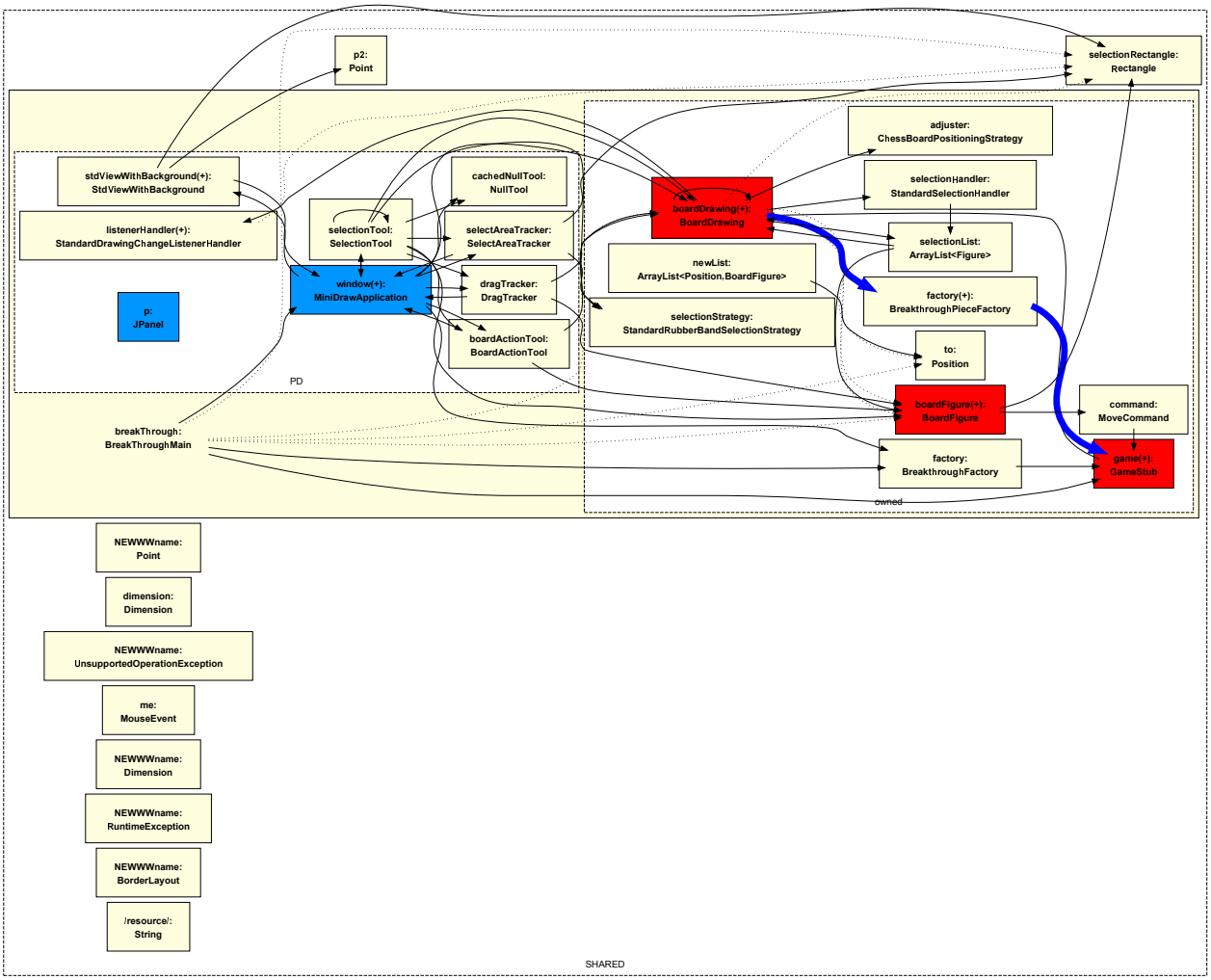


Figure 5.6: The top-level domains of the object graph for MD using the Assisted mode. The root object is expanded. The (+) on an object label indicates a collapsed object sub-structure.

2. If the association relation is one-to-one from class A to B, then the object of type A can be part of the object of type B (MakePartOf) or they can be peers (MakePeerWith);
3. If the association is one-to-many from object of type A to the object of type B, there is a collection of objects of type B and ideally the collection is strictly encapsulated in the object of type A, if A does not expose an alias to the collection;
4. An aggregation relation (has-a) indicates either a MakePeerWith or a MakePartOf refinement between the corresponding objects;
5. For an interface or abstract class in a class diagram, we find objects of the concrete classes or subclasses as the source and destination objects of a refinement.

5.4.4 Details of the Identified Refinements

By applying the procedure of finding refinements from the class diagrams and expressing the Model-View design, we found the following refinements to express the design intent of MD. The summary of the refinements in shown in Table 5.7.

- **MakePartOf-TLD**(`MiniDrawApplication`, `BreakThrough`): This refinement expresses the two-tiered Model-View design; the object of type `MiniDrawApplication` is a View object, since it is a subtype of the Drawing interface; we move this object to the top-level `PD` domain;
- **MakePartOf-TLD**(`SelectionTool`, `BreakThrough`): This refinement moves the object of type `SelectionTool` to the top-level domain `PD` of the root object to express the two-tiered Model-View design;
- **MakeOwnedBy-TLD**(`BoardDrawing`, `BreakThrough`): This refinement expresses that the object of type `BoardDrawing` that is a Model object and it is in the top-level `owned` domain; the refinement is skipped because the object is created by the factory object `BreakthroughFactory`; so we apply another refinement on the factory object to move it to the top-level `owned` domain;
- **MakeOwnedBy-TLD**(`BoardFigure`, `BreakThrough`): Another refinement is for moving a Model object into the top-level `owned` domain, but the `BoardFigure` object is also created by a factory object (`BreakthroughPieceFactory`), so the refinement is skipped;
- **MakeOwnedBy-TLD**(`BreakthroughFactory`, `BreakThrough`): This refinement moves a factory object to the top-level domain `owned`; the factories create Model objects; in order to place the factory in the same domain as the object that it creates (`BoardDrawing`), we move the factory in the top-level `owned`;
- **MakePartOf-TLD**(`JPanel`, `BreakThrough`): The object of type `JPanel` is also a View object and we move it to the top-level `PD`;
- **MakeOwnedBy-TLD**(`GameStub`, `BreakThrough`): In the class diagram for the

`breakthough` package, there is an association between `Game` and `BreakThoughMain` classes; since Game is an interface type and `GameStub` is its subclass, we do this refinement on the object of type `GameStub` to move this model object to the top-level `owned` domain;

- **MakeOwnedBy-TLD**(`BreakthroughPieceFactory`, `BreakThrough`): This refinement moves another factory object to the top-level domain `owned` to make the factory object peer with the object that it creates (`BoardFigure`);
- **MakeOwnedBy**(`ArrayList<Figure>`, `BoardDrawing`): In the class diagram for the `boardgame` package, there is a one-to-many association from `BoardDrawing` to `BoardFigure`; since the association is one-to-many, we understand that the `BoardDrawing` class have a collection of `BoardFifure` objects and apply `MakeOwnedBy` to make the collection strictly encapsulated;
- **MakePeerWith**(`BoardActionTool`, `BoardFigure`): The class diagram of the `boardgame` package shows an association between `BoardActionTool` and `BoardFigure` classes; `BoardActionTool` is a tool class provided by the framework, which provide some functionality to be used by the `BoardFigure` class;
- **MakePeerWith**(`MoveCommand`, `BoardFigure`): `BoardFigure` creates the `Command` object to implement the Command design pattern e.g., to enable the undo functionality; so there is an association between the two classes, but no object would own the Command object; we apply this refinement to make the object of type `MoveCommand` peers the object of type `BoardFigure`;
- **MakePeerWith**(`PositioningStrategy`, `BoardFigure`): In the `boardgame` package class diagram, `PositioningStrategy` implements the Strategy design pattern; some algorithms of `BoardFigure` is represented in the Strategy object e.g., to enable finding a position calculations; we make the Strategy object peer with `BoardFigure` by doing a MakePeerWith refinement;
- **MakePeerWith**(`DrawingChangeEvent`, `BoardDrawing`): In the class diagram for

Table 5.7: Refinements on MD to express design intent. A "-TLD" after the type of a refinement means that the destination domain is a domain of the root object.

No.	Ref.	Source Type	Destination Type	#Mins	Status
1	MPO-TLD	`MiniDrawApplication`	`BreakThrough`	0.87	Done
2	MPO-TLD	`SelectionTool`	`BreakThrough`	0.17	Done
3	MOB-TLD	`BoardDrawing`	`BreakThrough`	0.04	Skipped
4	MOB-TLD	`BoardFigure`	`BreakThrough`	0.04	Skipped
5	MOB-TLD	`BreakthroughFactory`	`BreakThrough`	0.11	Done
6	MPO-TLD	`JPanel`	`BreakThrough`	0.14	Done
7	MOB-TLD	`GameStub`	`BreakThrough`	_	Done (auto-ref)
8	MOB-TLD	`BreakthroughPieceFactory`	`BreakThrough`	_	Done (auto-ref)
9	MOB	`ArrayList<Figure>`	`BoardDrawing`	_	Not Attempted
10	MPW	`BoardActionTool`	`BoardFigure`	_	Not Attempted
11	MPW	`MoveCommand`	`BoardFigure`	_	Done (auto-ref)
12	MPW	`PositioningStrategy`	`BoardFigure`	_	Done (auto-ref)
13	MPW	`FigureChangeEvent`	`BoardFigure`	_	Done (auto-ref)
14	MPW	`DrawingChangeEvent`	`BoardDrawing`	_	Done (auto-ref)

Table 5.8: Summary of the manual refinements on MD to express design intent.

#Refinements	#Attempted	#Done	#Not-Attempted	#Auto-Ref	#Mins (Avg)
14	6	4	8	6	0.22

the `framework` package, there is a one-to-one association between the types `DrawingChangeEvent` and `Drawing`; so we apply a MakePeerWith refinement between the objects of types `DrawingChangeEvent` and `BoardDrawing`.

- **MakePeerWith**(`FigureChangeEvent`, `BoardFigure`): In the class diagram of the `framework` package, there is a one-to-one association between the types `FigureChangeEvent` and `Figure`; so we apply another MakePeerWith refinement between `FigureChangeEvent` and `BoardFigure` as the subclass of `Figure`.

Summary of the refinements. Table 5.8 shows the summary of the refinements. We identify 14 refinements to attempt, attempt 6 refinements, and of those, 4 are done. In the process of attempting the first 6 refinements, another 6 are done as auto-refinements, which we explain in Section 3.5. The details of the refinements are in Section 5.4.4.

After we attempt refinements to express the design intent, OOGRE cannot find a typing, because there are conflicts to resolve. We could attempt more manual refinements to resolve conflicts, but for MD, it is harder since the system uses a framework to implement an application and there can be conflicts in parts of the framework code that are not used by

the application. In that case, we cannot do more refinements to resolve conflicts, since there may be no object in the object graph for which the traceability information includes variables involved in the conflict expression. So to find a typing, we launch the Assisted mode and let OOGRE attempt more refinements on variables (as opposed to objects in the graph).

5.4.5 Metrics on the Inferred Qualifiers

The qualifiers in the typings inferred using the Auto and Assisted modes are shown in Table 5.9. By looking at the numbers, we confirm that the Auto mode tends to respect the ranking more by inferring more `owned`, `PD` and `owner` qualifiers. The Assisted mode infers more `p` and `shared` qualifiers, because of the manual refinements that we apply to infer the two-tiered architecture. Our design intent is to split the top-level objects across two top-level domains, so the analysis infers 50 qualifiers with `p` as the owning domain. The number of `lent` and `unique` qualifiers is almost the same for Auto and Assisted modes.

Table 5.9: Number of different qualifiers in the inferred typings using Auto and Assisted modes. The first column indicates the number of variables. The columns with a modifier indicate the number of qualifiers with that modifier as the owning domain in the typing. For example, the column `owned` indicates the number of qualifiers in the form of <`owned`, q> where q can be any modifier. We have a column for the number of qualifiers in the form of <`p`>.

Experiment	**vars**	`owned`	`PD`	`owner`	`p`	`shared`	<`p`>	`lent`	`unique`
Auto	521	40	16	167	14	129	75	55	25
Assisted	521	35	11	138	50	140	75	49	23

5.4.6 Intent: Expressing Two-Tiered Design

For MD, we study the documentation and propose refinements to express our design intent, and OOGRE finds a typing and extracts a refined object graph. We confirm that the extracted object graph visually expresses the design intent from the class diagrams. The main design intent in MD is the two-tiered Model-View architecture. The `MiniDraw` and `BoardGame` frameworks follow that style and impose it on the client applications. We use the two top-level domains `owned` and `PD` to express Model and View, respectively. Therefore, we place `DrawingView` and its subclasses like `MiniDrawApplication` that are View objects in the top-level `PD` and Model objects like `Game`, `BoardDrawing` and `BoardFigure` in the top-level `owned`.

Expressing Model. The Model objects like `BoardDrawing` and `BoardFigure` are created by some factory objects, so are not able to move them to the top-level `owned` directly using MakeOwnedBy. We apply the refinement on the factory object to move the objects that the factory creates to the same domain. By studying the object graph, we confirm that the Model objects are in the top-level `owned`. Some of the remaining refinements are done as auto-refinements. For example, we want to place the object of type `GameStub` in Model. We get the result as an auto-refinement (#7 in Table 5.7, Section 5.4.4). Also we want to express that the objects of types `FigureChangeEvent` and `DrawingChangeEvent` are peers with the objects of type `BoardFigure` and `BoardDrawing`, respectively. The auto-refinements (13 and 14 in Table 5.7) achieve this result.

Expressing View. We apply refinements to place objects such as `JPanel` in View. We apply a MakePartOf on the object of type `JPanel` and move it to the top-level `PD` domain. Some other View objects like the object of type `StandardDrawingChangeListenerHandler` also move to the top-level `PD` due to auto-refinements. By looking at the object graph, we confirm that View objects are in the top-level `PD` domain. Overall, we are able to use refinements to make the object graph reflect our design intent.

5.4.7 Object Graphs

We show the object graph of MD that is obtained from the Assisted mode in Fig. 5.6. By applying manual refinements, the Assisted mode enables us to express the two-tiered design, Model-View. The View objects such as the objects of type `MiniDrawApplication`, `JPanel` and subclasses of `Tool` are in the top-level `PD` domain. Model objects such as BoardDrawing and BoardFigure are in the top-level `owned` domain. The Auto mode does not distinguish between Model and View objects and moves all the top-level objects to the top-level domain `owned` (Fig. 5.5). The reason is Auto attempts first MakeOwnedBy and the refinements on those objects are done.

5.4.8 Analysis of the Object Graphs

We show the flat object graph of MD in Fig. 5.7. The top-level of the object graphs from the Auto and Assisted modes are less busy comparing the flat one. The object graph from the Auto mode adds hierarchy to the flat object graph, but is does not express the design intent of the application. So we use the Assisted mode to get a hierarchical object graph that expresses our design intent.

By comparing the hierarchical object graphs from the Auto and Assisted modes, we notice that the Assisted mode expresses the Model-View design of MD by splitting the top-level objects between the two top-level domains. In the object graph from the Auto mode, all the top-level objects are in the same top-level domains `owned`. Using Assisted, the object graph reflects our design intent, but in the resulting object graph for Auto, all the objects are placed in the top-level owned domain, which does not express the two-tier design.

In the object graph from the Assisted mode, the objects that contain the logic of the application, such as `GameStub`, `BoardDrawing` and `BoardFigure` (highlighted in the object graph, Fig. 5.6) are in the top-level domain `owned` that is mapped to Model. The user interface objects, such as `swing` objects, and subtypes of `Drawing` interface (highlighted in the object graph) are in the top-level domain `PD`, which is mapped to View. Some objects are in the `shared` domain, which can be due to using manifest ownership e.g., objects of type `String`. Also, as we mentioned for JDepend, if we show other kinds of edges, then we would see less objects with no edges. We show only the points-to edges in the object graphs.

5.4.9 Metrics on the Object Graphs

We compute some metrics on the object graphs of the Auto and Assisted modes (Table 5.10). The object graph of the Auto mode has higher #O, since it distinguished more objects by applying the automated refinements on each field and local variable. The Assisted mode has higher #TLO, because some of our manual refinements make some objects peers, such as the factory objects and the objects that they create. Expressing the two-tiered de-

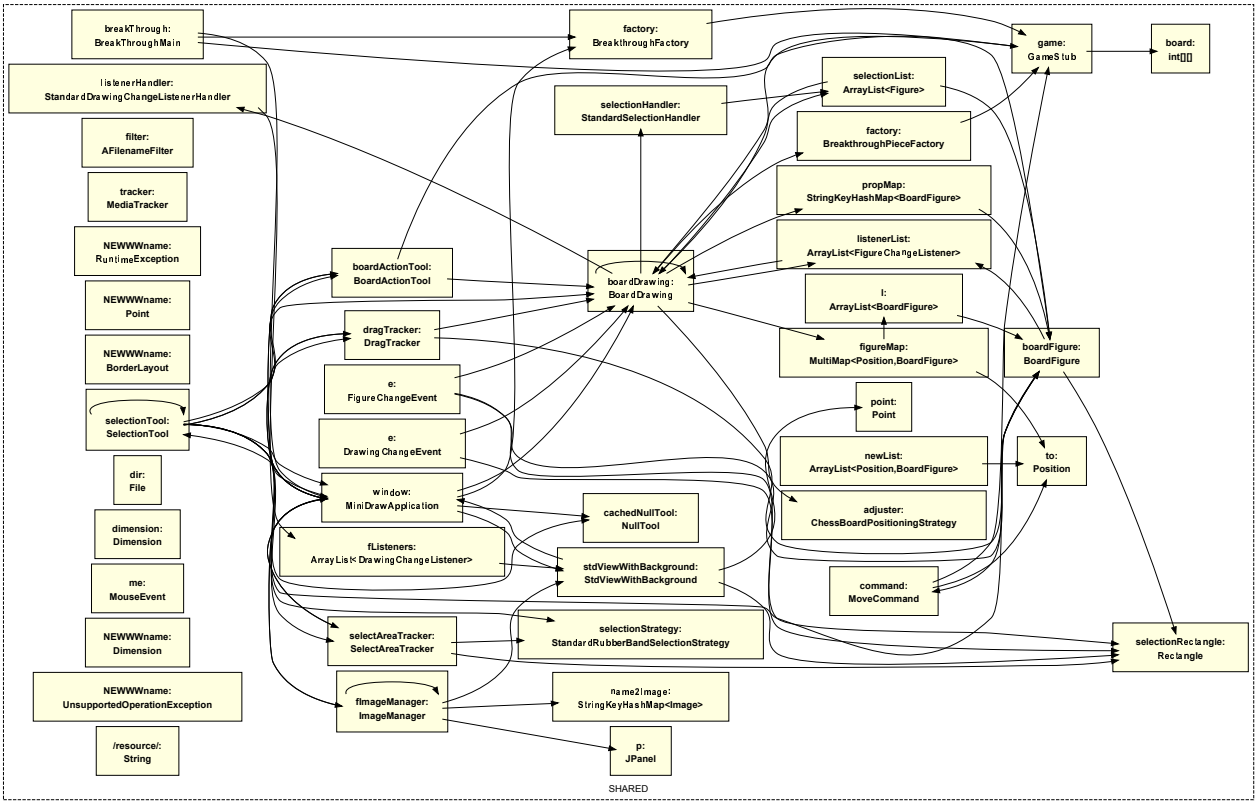


Figure 5.7: The flat object graph extracted from the default qualifiers for MD.

sign using Assisted mode leads to higher number of objects in `PD` (#OPD). However, the Auto mode has more objects in `owned`, since it applies MakeOwnedBy refinements first. The depth of hierarchy in both object graphs is almost the same, but MAX OD for the Auto mode is higher, due to all the automated refinements that the Auto mode applies.

Table 5.10: Object graph metrics extracted from the inferred typings across Auto and Assisted modes.

Experiment	**#O**	**#TLO**	**#LLO**	**#OPD**	**#OPrD**	**MXD**	**AOD**	**min OD**	**max OD**
Auto	57	19	1	6	50	4	0.53	0	19
Assisted	47	21	1	9	37	4	0.53	0	12

CHAPTER 6 DISCUSSION, LIMITATIONS, FUTURE WORK AND CONCLUSION

In this chapter, first we revisit each hypothesis and and summarize our evidence for each one. Then we revisit the thesis statement of this dissertation to see how the approach meets its goals overall. Next, we mention some limitations of the approach, some future work and conclude.

6.1 Hypotheses Revisited

We revisit the hypotheses, their success criteria and evidence, with respect to the results of evaluating OOGRE on the subject systems JDepend and MD. We provide specific examples on how each success criteria or evidence is supported by the result of our evaluation.

H1: Hierarchical Object Graph

A hierarchical object graph leads to fewer visible objects at the higher levels of the object graph, since more objects are collapsed underneath other objects, compared to a flat object graph, in which all the objects are at the same hierarchy level.

Success criteria.

- Using OOGRE, we are able to extract hierarchical object graphs for the JDepend and MD subject systems, that contain two types of hierarchy, strict encapsulation that is illustrated using MakeOwnedBy refinements and logical containment that is the result of MakePartOf refinements;
- Using OOGRE, we express an object hierarchy that provides architectural abstraction. For example, in Table 5.2, using the refinement number 10, we move an object of type `ArrayList<String>`, which is a data structure to lower levels of the object graph;
- Using the SplitUp refinement, we split an object into objects of the same type in

different domains, so the object graph is domain-sensitive, since the objects of the same type are placed in different domains. For example, in Fig. 5.2, distinct objects of the type `ArrayList<String>` are placed the `owned` domains of distinct objects.

Evidence.

- For JDepend and MD, we compute metrics on the hierarchical and flat object graphs. For JDepend, the number of top-level objects for the Auto and Manual experiment are 5 and 18 respectively, while it is 26 for the OT-repro experiment. This shows that the object graph of the SOD experiments have less busy top-level domains. As another example, The maximum object depth (max OD) of the object graph of Auto is 36, while it is 26 for OT-repro. The max OD metric for Manual is 18, since we express our design intent that is not only gaining more hierarchy. For MD, the max OD numbers of the Auto and Assisted are 19 and 12, respectively, which is higher than the max OD of the flat object graph. So we confirm that a hierarchical object graph has a lower number of objects at its top-level domains.

H2: Logical Containment

Applying both strict encapsulation and logical containment that are supported by SOD, compared to other type systems that support only strict encapsulation but no logical containment, leads to a more hierarchical object graph. A strictly encapsulated object is dominated by another object and all accesses to it must go through its dominator. A logically contained object is part-of another object, but still accessible to other objects.

Success criteria.

- For MD, there are 6 and 5 done MakePartOf refinements for Auto and Assisted, respectively. For JDepend, we attempts 4 manual MakePartOf refinements and for Auto, there are 109 attempted MakePartOf refinements out of which 29 are done. So we confirm that OOGRE infers `PD` qualifiers. We also attempt MakePartOf refinements to express design intent. For example, in Table 5.2, the refinement number 16 is to express the Composite design pattern by making the object of type `PackageComparator`

logically contained in the object of the same type;

- Edges of the object graphs indicate that the objects that are in the public domain `PD`, can be accessed by other objects. For example, in Fig. 5.1 the object of type `JDepend` can access the object of type `ClassFileParser` that is in the `PD` domain;
- Without supporting logical containment in a hierarchical object graph, a lower level object stays as peer of its intended parent object in order to remain accessible to other objects. For example, for JDepend, the OT-repro experiment has 26 objects in the top-level domains, while the number of objects in top-level domains for the Auto experiment is 5;
- Object graphs of each subject system illustrate the notion of logical containment that groups objects in public domains of the other objects. For example, for JDepend, there are 82 objects public domains for Auto and there are 48 objects in public domains for Manual. For MD, there are 50 objects in public domains for Auto and 37 objects for Assisted.

Evidence.

- In Table 5.3, we show the number of inferred qualifiers for the three experiments on JDepend. Having 59 and 33 variables with `PD` qualifiers for Auto and Manual confirm the number of objects in a public domain in Table 5.5. The number of inferred qualifiers for Auto and Assisted for MD is 16 and 11, respectively.

H3: Object Graphs vs. Qualifiers

Using an interactive inference tool for SOD, developers refine an object graph to express their design intent by make owned by, make part of, make peer with and make shared refinements. Behind the scene, a static analysis infers valid qualifiers that type-check and the refinement is consider done. Otherwise the refinement is skipped and the qualifiers in the code and the graph are unchanged.

Success criteria.

- During the evaluation of JDepend and MD, we are able to obtain a hierarchical object

graphs that express our design intent, only by doing refinements and not by modifying the qualifiers. In Table 5.4, we show that for the Manual experiment of JDepend, we do 28 refinement, and for OT-repro we add 21 qualifiers manually. However, adding a qualifier manually involves adding both the owning domain and the domain parameter. A refinement indicates only the owning domain.

- Some of the refinements we perform express our design intent. We identify those refinements in the summary tables using (D). We do some additional refinements to resolve conflicts. We identify those refinements using a (C) (Table 5.2);
- Each refinement that we apply preserves the results of all the previous done refinements, and does not undo them. As a result, some refinements can be skipped because of some previous done refinements. This is the case for skipped MakePeerWith refinements of both subject systems using Auto or Assisted.
- our results show that the skipped refinements do not change the object graph and it stays unchanged, since the analysis does not infer new qualifiers.

Evidence.

- We implement the adaptation and transfer functions for the full Java language, including classes, interfaces, inheritance (method overriding), generics types, etc. The main language features we do not support are static fields and methods, arrays, among others.
- We evaluate our implementation on one small subject system (MD, 1.4 KLOC) and one mid-size subject system (JDepend, 5KLOC); MD has design design documentation that we mine for the design intent; JDepend has data from a previous experiment by Huang et al. which enables comparison;
- After obtaining the results on each subject system, we run the independent type-checker to validate the inferred qualifiers. There is no major type-checker warnings in the inferred qualifiers of none of the subject systems.
- By comparing the inferred qualifiers by OOGRE with the qualifiers that are inferred

by Huang el al., we confirm that by inferring qualifiers with as the owning domain `PD` (Table 5.3), OOGRE creates object graphs with more hierarchy.

- We compare the qualifiers across multiple type systems, to measure the difference in the hierarchy in the object graph, as well as the precision of the inferred qualifier. E.g., unlike the nearly hundred of any qualifiers inferred by UT for JDepend, we infer more precise ones.
- We show that the resulting object graphs convey the design intent of the documented class diagrams. For example, for MD, we are able to express Model-View design. When we collapse all the objects in the top-level domains, we obtain a nice diagram of the MD object structure.

H4: Inferring SOD Qualifiers

To express strict encapsulation and logical containment, the inference analysis constructs a set mapping that contains feasible qualifiers and multiple valid typings. Some of the feasible qualifiers contain ***`n`****`.PD`, where* ***`n`*** *is the name of an object, and* `PD` *is a public domain declared on the class of that object.*

Success criteria.

- On JDepend, we do 12 MakeOwnedBy refinements using the Manual experiment, and on MD we attempt 3 MakeOwnedBy refinements. The analysis infers qualifiers with `owned` as the owning domain for the target variables of the MakeOwnedBy refinements.
- Our results confirm that each object that is strictly encapsulated in the OT-repro experiment of JDepend, can also be encapsulated using SOD. If there is an object that is not encapsulated using SOD, that is our design intent to not make it strictly encapsulated. For example, the object of type Analyzer is strictly encapsulated in the Auto experiment of JDepend, but in the Manual experiment, we make it peers with the object of type `JDepend`.
- The inferred qualifiers in Table 5.3 and Table 5.9 confirm that OOGRE is able to infer `PD` for the MakePartOf refinements (manual or automated). The reason that there is

no `n.PD` in the inferred qualifiers is the missing `final` fields. In that case, OOGRE infers `lent` and the extraction analysis resolves `lent` into a valid domain in the object graph that corresponds to `n.PD`.

- Although the analysis infers `unique` and `lent`, but the extraction analysis resolves them to precise domains, so all the objects in the resulting object graph of each text case are in precise domains (`owned` or `PD`).

Evidence.

- In our implementation, and for a MakePartOf refinement, the destination object must create the source object, otherwise the refinement is not allowed.
- Most of the skipped refinements of each subject system, are due to unwanted aliasing for MakeOwnedBy refinement and missing `final` fields for the MakePartOf refinements;
- If a MakePeerWith refinement is skipped, it means a previous refinement causes it, since a qualifier with `owner` as the owning domain should work as a default qualifier.
- We do not measure the situation that the analysis tries to infer a precise qualifier, but the refinement is skipped because of some missing `final` fields. It would be good to measure the number of skipped MakePartOf refinements due to missing `final` fields.

H5: Finding a Valid Typing

The inference analysis finds a typing from the set mapping using the maximal qualifiers for each variable. If the typing is not a valid typing, developers keep doing refinements until the analysis finds a maximal typing that type-checks the program. If the analysis cannot do so, developers start over with another sequence of refinements.

Success criteria.

- After each refinement, when the analysis reaches the fixed point without any empty sets of qualifiers in the set mapping, it means there are multiple valid typings that satisfy all the done refinements.
- We manually check that all the target variables are in the destination domains of the applied refinements;

- For JDepend, we first attempt 16 refinements to express our design intent, but those are not enough, so we attempt 14 more refinements to resolve conflicts (Table 5.2). Only after attempting these 28 refinements, the analysis is able to find a valid maximal typing.
- For MD, we attempt 6 manual refinements and then we use the Assisted mode. After apply refinement on more than hundred variables, the analysis finds a valid maximal typing (Table 5.6).

Evidence.

- For JDepend, we apply 28 refinements out of which 24 are done and the optimality property holds after that. Table 5.1 shows that using Auto, less than half of the attempted refinements are completed. This is unsurprising since Auto attempts different refinements of the same variables.
- For MD, we use the Auto and Assisted modes. Table 5.6 confirms that also for MD, less than half of the automated refinements are done. For assisted, due to the constraints that the manual refinements impose, the number of done refinements is almost one third of the attempted ones;
- On JDepend and using the Manual mode, we attempt 28 refinements, but for OT-repro we add 21 qualifiers. Adding a qualifier manually involves adding both the owning domain and domain parameter. By attempting a refinement, we only indicate the owning domain.

H6: Ranking Qualifiers

The analysis ranks qualifiers that contain `unique` *and* `lent` *highest, followed by local domains (*`owned` *or* `PD`*), followed by domain parameters (*`owner` *or* `p`*). The lowest rank is the global domain (*`shared`*).*

Success criteria.

- Inferring qualifiers containing `lent` and `unique`, which are the highest ranked modifiers helps the analysis to utilize set-based solution for SOD efficiently. Previous versions

of the analysis without `lent` and `unique` do not work for a subject system as big as JDepend;

- We rank local domains (`owned` and `PD`) higher than domain parameters (`owner` and `p`). This leads to having higher total number of local domains compared to domain parameters for Auto in Table 5.3. The reason is Auto attempts MakeOwnedBy and MakePartOf refinements before MakePeerWith refinements. For Manual, the total number of qualifiers with a local domain as the owning domain is roughly the same as the total number of qualifiers with a domain parameter as the owning domain. For OT experiments (OT-repro and OT-orig), total number of domain parameters are significantly higher, due to not supporting public domains that leads to object graphs with less hierarchy;
- Table 5.3 confirms that for SOD experiments of JDepend, less than 30% of the inferred qualifiers are `shared`. Also Table 5.9 confirms that for MD, about 25% of the inferred qualifiers are `shared`.

Evidence.

- The metrics on object graphs (Table 5.5 for JDepend, and Table 5.10 for MD) confirm the approximation of their hierarchy by computing metrics on the inferred qualifiers (Table 5.3 for JDepend, and Table 5.9 for MD). For example, for JDepend, Auto infers more local domains compare to Manual, so the approximation is the object graph from Auto should be more hierarchical. Table 5.5 confirms that the object graph from the Auto experiment is more hierarchical with max OD equal to 36 compared to the object graph from the Manual with max OD equal to 18.

H7: Automation

The tool supports two additional modes for applying refinements, Auto, a fully automated mode when the developers do not do any manual refinements; and Assisted, once the developers finish applying refinements and they just want the tool to find a typing.

Success criteria.

- We run the Auto mode on JDepend and MD and OOGRE is able to find a typing and extracts a hierarchical object graph without any additional input;
  - We use the Assisted mode on MD and OOGRE is able to find a typing and extracts a hierarchical object graph by respecting some previously attempted manual refinements;
  - The Auto and Assisted modes make an object logically contained if it cannot be strictly encapsulated and make the object peer with other objects if it cannot be logically contained. The number of MakePeerWith (MPW) refinements in Table 5.1 and Table 5.6 show the number of variables that are neither strictly encapsulated nor logically contained;
- The Assisted mode that is applied on MD, respects the done manual refinements attempted on the subject system.

Evidence.

- We measure the percentages of done auto-refinements of each type;
- We measure the portion of done auto-refinements out of the attempted ones;

Thesis Statement Revisited

Using our approach, developers are able to refine an object graph directly using the supported refinements. They make an object strictly encapsulated using MakeOwnedBy, logically contained using MakePartOf, peer with another object using MakePeerWith, or globally aliased using MakeShared. Then the inference analysis infers qualifiers that type check, if the code as written supports the requested refinement. The refined hierarchical object graph that applies abstraction by hierarchy and by type expresses the design intent of the developers. Our approach is a good step, but suffers from some limitations, which we discuss in the next section.

Scaling the process of extracting and refining object graphs. While we agree that 1hr/5KLOC is slow (JDepend subject system), it is still much better than the manual effort, estimated to be at 1hr/1KLOC by Abi-Antoun et al., [6], just to get an initial object

graph and not including the time spent refining the object graph. We have not particularly optimized the implementation for ease of debugging. We think the tool can be sped up by reducing the amount of copying of sets of qualifiers. Our initial profiling points to the set creation and copying as being a hotspot in the execution. Only for UT is the inference quadratic but it infers many `any` qualifiers that do not convey precise information about the ownership context. For OT and SOD, every refinement requires quadratic time, so running the Auto mode on all the variables is cubic in terms of the number of variables in the program.

To make the Auto mode faster, we need smarter ways of resolving conflicts. For example, the Auto mode attempts many refinements. Some are necessary, some are not. Our current implementation of Auto mode is brute-force. A better solution would be to have the tool work more like the Manual Mode, i.e., analyze the remaining conflicts (just like in the MoreInfoNeeded window) and focus only on resolving conflicts, i.e., propose refinements that only touch the variables that are still in conflict, as opposed to all possible refinements. Moreover, it may be helpful to apply some weighting strategy to pick a variable to apply a refinement on. For example, it could be useful to lock down the sets of qualifiers of fields, before those of method parameters or returns. Strategy is to use the size of the set of qualifiers of a variable. Applying refinements on a variable with a bigger set of qualifiers enforces more constraints on the set mapping so the set-based solution may reach a fixed point faster.

6.2 Limitations

In this section, we organize the limitations of this work in two groups: first, approach limitations resulting from our design decisions; and second, type system limitations that are due to using SOD as our underlying type system.

6.2.1 Approach Limitations

No typing, no object graph. Ideally, and for being interactive, the approach should show a refined object graph to developers, after each refinement. Since for a program with an arbitrary set of qualifiers, the optimality property may not hold, our approach may not find a typing after each refinement and developers may have to do more refinements to resolve any conflicts. Only after developers resolve the remaining conflicts can the approach find a valid typing and extract the refined object graph. This way, developers see the result of their refinements all at once on the object graph, which is perhaps less usable than seeing the effect of each refinement, one at a time. Moreover, since each refinement is applied on the result of previous done refinements, it is more helpful to developers to see on what object graph they are attempting the next refinement.

Respecting auto-refinements. After each done refinement, OOGRE not only respects the changes by the requested refinement in the set mapping, but also the side effects of the requested refinement (auto-refinements). This design decision may lead to a refinement being skipped if it contradicts a previous, done auto-refinement. This arises when developers attempt a refinement where the source object is used in an auto-refinement. Auto-refinements do not make the sets of the target variables to be singletons. They pick only the highest ranked qualifier from the set of qualifiers of the target variables. By doing that, auto-refinement choose the best possible answer, which the developers may not have picked if they were to do a manual refinement. If the developers do not want the inferred qualifier (the highest ranked one), they would do a refinement that infers a lower ranked qualifier, and that qualifier would be in the set of qualifiers of the target variables.

Hard-coded number of domains and hard-coded domain names. By hard-coding the number of domains to two (one private and one public), developers cannot express certain designs like Model-View-Controller that needs three top-level domains. By hard-coding the domain names to be `owned`, `PD`, and `p`, we enable the analysis to enumerate and infer them,

but the design intent of the domain is not obvious from its name. As a result, developers are not able to define multiple domains per class, or to have the domain names express design intent.

A refinement to move an object into the domain parameter `p`. We are not sure if we need a type of a refinement that moves an object into the domain parameter `p`. We do not support this refinement for two reasons. First, due to SOD constraints, an object cannot be created inside a domain parameter. So the owning domain of the qualifier on the left hand side of an object creation expression cannot be `p`. Second, the domain parameter `p` binds to other domains, so the developers may not know the result of their refinement on the object graphs. However, this kind of refinement makes sense at the level of variables during the Auto and Assisted modes.

One ownership parameter. The approach currently supports the implicit `owner` and the explicit `p` ownership parameter. We support a single domain parameter to keep the inference tractable, but this reduces expressiveness. Some data structures and programming idioms, require more than one explicit ownership parameter. For example, to express a standard hashtable, two ownership parameters are needed: one for the key objects and one for the value objects [12]. SOD can still express hashtable with one parameter, by making the hashtable itself, the key object, or the value object be peers. As a workaround, we can also specialize a hashtable to have a specific key type or value type and reserve the parameter for the more interesting objects (See Section 3.10.5).

6.2.2 Type System Limitations

Final fields. In SOD, the code can refer to the public domain of an object through a final field `n`, using the construct `n.PD`. If the variable `n` is not final, it may be re-assigned, and the type system would lose track of the relationship between an object and the objects contained in its public domain. In other ownership type systems, where `n` is always `this`, this is not an issue. In SOD, it also possible to refer to a public domain through a sequence of final fields `n1.n2...PD`, though that is not part of our formalization. The adaptation and the inference

analysis also introduce these qualifiers. If `n` is not final, then the adaptation functions infer `lent` instead of `n.PD`, as long as it does not flow to a qualifier that is not `lent`.

Using manifest ownership for `String`'s. In the current version of the analysis, we use manifest ownership to infer the qualifier <`shared`, `shared`> for the variables of type String. Although this simplifies the analysis work to find a valid typing, for reasoning about security, we may need to reason more carefully about objects of type `String`. Ideally, the analysis should treat objects of type `String` like other objects, i. e., by assigning them the initial set of qualifiers.

Existential domain. The modifiers `unique` and `lent` add some amount of expressiveness to SOD, but their usage is restricted. For example, `lent` is not allowed on the field variables. What would increase the SOD type system's expressiveness is some variant of an existential domain that type-checks many expressions, and can be resolved to a precise domain using a separate static analysis. UT supports its own variant of an existential qualifier, `any` (really, means readonly), that does not indicate the actual ownership information of an object, so it type-checks many expressions, but requires additional purity qualifiers on methods to indicate that they do not mutate any objects. The UT qualifier `any` can be used on fields among other variables, and enables the optimality property to hold for any program with arbitrary qualifiers.

6.3 Future Work

In this section, we mention possible future work.

Partial qualifiers. It would be useful to allow developers to add some partial qualifiers for some of the variables and have analysis read the qualifiers and use them. This feature is available in many inference tools [17, 22, 21].

Conduct a user study. We will evaluate the WYSIWYG claim and the visual aspect of our approach by conducting a user study. One study design involves having one group of participants use the tool to express their design intent while the participants of other group

add the qualifiers manually. We will measure if by using the tool, the participants refine the object graph to reflect their design intent faster. An alternative design is to have the participants use other type systems such as OT or UT. Another study design may measure the learnability of the tool by teaching Ownership Domains to novice developers.

6.4 Conclusion

We propose, implement and evaluate an approach where developers express their design intent by refining an object graph directly, while an analysis infers valid ownership type qualifiers in the code. These qualifiers are used by a separate extraction analysis to extract an updated graph. Automating the process of extracting and refining object graphs makes their applications more practical and more readily adoptable by developers.

REFERENCES

[1] The Crystal Static Analysis Framework, 2015. http://code.google.com/p/crystalsaf.

[2] The ObjectAid UML Explorer for Eclipse, 2016. http://www.objectaid.com.

[3] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, CMU, 2010.

[4] M. Abi-Antoun and J. Aldrich. Ownership Domains in the Real World. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 93–104, 2007.

[5] M. Abi-Antoun and J. Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 321–340, 2009.

[6] M. Abi-Antoun, N. Ammar, and Z. Hailat. Extraction of Ownership Object Graphs from Object-Oriented Code: an Experience Report. In *International Conference Series on the Quality of Software Architectures (QoSA)*, 133–142, 2012.

[7] M. Abi-Antoun and J. M. Barnes. Analyzing Security Architectures. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 3–12, 2010.

[8] M. Abi-Antoun, A. Giang, S. Chandrashekar, and E. Khalaj. The Eclipse Runtime Perspective for Object-oriented Code Exploration and Program Comprehension. In *Proceedings of the 2014 Workshop on Eclipse Technology eXchange*, ETX '14, 3–8, 2014.

[9] M. Abi-Antoun, E. Khalaj, R. Vanciu, and A. Moghimi. Abstract Runtime Structure for Reasoning About Security: Poster. In *Proceedings of the Symposium and Bootcamp on the Science of Security*, HotSos '16, 1–3, 2016.

[10] M. Abi-Antoun, Y. Wang, E. Khalaj, A. Giang, and V. Rajlich. Impact analysis based on a global hierarchical Object Graph. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 221–230, 2015.

[11] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, 1–25, 2004.

[12] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 311–330, 2002.

[13] N. Ammar and M. Abi-Antoun. Empirical Evaluation of Diagrams of the Run-time Structure for Coding Tasks. In *Working Conference on Reverse Engineering (WCRE)*, 367–376, 2012.

[14] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[15] D. Clarke, J. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 48–64, 1998.

[16] W. Dietl, S. Dietzel, M. Ernst, K. Muslu, and T. Schiller. Building and using pluggable type-checkers. In *International Conference on Software Engineering (ICSE)*, 681–690, 2011.

[17] W. Dietl, M. Ernst, and P. Müller. Tunable Static Inference for Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, 333–357, 2011.

[18] W. Dietl and P. Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[19] C. Dymnikov, D. J. Pearce, and A. Potanin. OwnKit: Inferring Modularly Checkable Ownership Annotations for Java. In *Australian Software Engineering Conference (ASWEC)*, 181–190, 2013.

[20] S. Hauck. APHYDS: The Academic Physical Design Skeleton. In *International Conference on Microelectronics Systems Education*, 8–, 2003.

[21] W. Huang, W. Dietl, A. Milanova, and M. Ernst. Inference and Checking of Object Ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, 181–

206, 2012.

[22] W. Huang and A. Milanova. Towards effective inference and checking of ownership types. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2011.

[23] M. E. Khalaj. Interactive Refinement of Hierarchical Object Graphs. Master's thesis, Department of Computer Science, Wayne State University, 2016.

[24] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. Abstracting Runtime Heaps for Program Understanding. *Transactions on Software Engineering (TSE)*, 39(6):774–786, 2013.

[25] A. Milanova and J. Vitek. Static Dominance Inference. In *International Conference on Objects, Models, Components, Patterns (TOOLS)*, 211–227, 2011.

[26] N. Mitchell. The Runtime Structure of Object Ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, 57–64, 2006.

[27] S. Nägeli. Ownership in Design Patterns. Master's thesis, Department of Computer Science, Federal Institute of Technology Zurich, 2006.

[28] D. Rayside and L. Mendel. Object Ownership Profiling: a Technique for Finding and Fixing Memory Leaks. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007.

[29] J. Schäfer and A. Poetzsch-Heffter. A Parameterized Type System for Simple Loose Ownership Domains. *Journal of Object Technology*, 6:71–100, 2007.

[30] A. Spiegel. *Automatic Distribution of Object-Oriented Programs.* PhD thesis, Freie Universität Berlin, 2002.

[31] M. Vakilian, A. Phaosawasdi, M. D. Ernst, and R. E. Johnson. Cascade: a universal programmer-assisted type qualifier inference tool. In *International Conference on Software Engineering (ICSE)*, 234–245, 2015.

[32] R. Vanciu. *Static Extraction of Dataflow Communication for Security.* PhD thesis, Wayne State University, 2014.

[33] R. Vanciu and M. Abi-Antoun. Ownership Object Graphs with Dataflow Edges. In *Working Conference on Reverse Engineering (WCRE)*, 267–276, 2012.

[34] R. Vanciu and M. Abi-Antoun. Finding Architectural Flaws Using Constraints. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 334–344, 2013.

[35] R. Vanciu and M. Abi-Antoun. Object Graphs with Ownership Domains: an Empirical Study. Springer LNCS 7850:109–155, 2013.

[36] D. Werner and P. Müller. Runtime Universe Type Inference. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2007.

[37] H. S. Zhu and Y. D. Liu. Heap Decomposition Inference with Linear Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 104–128, 2013.

ABSTRACT

AUTOMATED REFINEMENT OF HIERARCHICAL OBJECT GRAPHS

by

MOHAMMAD EBRAHIM KHALAJ

May 2017

Advisor: Dr. Marwan Abi-Antoun

Major: Computer Science

Degree: Doctor of Philosophy

Object graphs help explain the runtime structure of a system. To make object graphs convey design intent, one insight is to use abstraction by hierarchy, i.e., to show objects that are implementation details as children of architecturally-relevant objects from the application domain. But additional information is needed to express this object hierarchy, using ownership type qualifiers in the code. Adding qualifiers after the fact involves manual overhead, and requires developers to switch between adding qualifiers in the code and looking at abstract object graphs to understand the object structures that the qualifiers describe.

We propose an approach where developers express their design intent by refining an object graph directly, while an inference analysis infers valid qualifiers in the code. We present, formalize and implement the inference analysis. Novel features of the inference analysis compared to closely related work include a larger set of qualifiers to support less restrictive object hierarchy (logical containment) in addition to strict hierarchy (strict encapsulation), as well as object uniqueness and object borrowing. A separate extraction analysis then uses these qualifiers and extracts an updated object graph.

We evaluate the approach on two subject systems. One of the subject systems is reproduced from an experiment using related techniques and another ownership type system, which enables a meaningful comparison. For the other subject system, we use its documenta-

tion to pick refinements that express design intent. We compute metrics on the refinements (how many attempts on each subject system) and classify them by their type. We also compute metrics on the inferred qualifiers and metrics on the object graphs to enable quantitative comparison. Moreover, we qualitatively compare the hierarchical object graphs with the flat object graphs and with each other, by highlighting how they express design intent. Finally, we confirm that the approach can infer from refinements valid qualifiers such that the extracted object graphs reflect the design intent of the refinements.

AUTOBIOGRAPHICAL STATEMENT

Mohammad Ebrahim Khalaj

Education

- M.S. Computer Science, Wayne State University, Detroit MI, USA, December 2016.
- M.S. Computer Software Engineering, Sharif University of Technology, Tehran, Iran, October 2011.
- B.S. Computer Software Engineering, Shahid Beheshti University, Tehran, Iran, November 2008.

Peer review publications

1. Abi-Antoun, M., Khalaj, E., Vanciu, R., and Moghimi, A. Abstract Runtime Structure for Reasoning about Security. *Poster at Symposium and Bootcamp on the Science of Security (HotSoS)* (2016).
2. Abi-Antoun, M., Wang, Y., Khalaj, E., Giang, A., and Rajlich, V Impact Analysis based on a Global Hierarchical Object Graph. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015).
3. Khalaj, E., Vanciu, R., and Abi-Antoun, M. Is There Value in Reasoning about Security at the Architectural Level: a Comparative Evaluation. *Poster at Symposium and Bootcamp on the Science of Security (HotSoS)* (2014).
4. Vanciu, R., Khalaj, E. and Abi-Antoun, M. Comparative Evaluation of Architectural and Code-Level Approaches for Finding Security Vulnerabilities. In *Workshop on Security Information Workers (SIW), co-located with the ACM Conference on Computer and Communications Security (CCS)* (2014).
5. Abi-Antoun, M., Giang, A., Chandrashekar, S., and Khalaj, E. The Eclipse Runtime Perspective for Object-Oriented Code Exploration and Program Comprehension. In *Eclipse Technology eXchange Workshop (ETX)* (2014).

Technical reports

- Khalaj E, Vanciu R. and Abi-Antoun M. Comparative Evaluation of Static Analyses that Find Security Vulnerabilities. SEVERE Technical Report, April 2014.

Master's thesis

- Khalaj, E. Interactive Refinement of Hierarchical Object Graphs, Master Thesis – Wayne State University, December 2016.